Proving the TLS Handshake Secure (as it is)

Karthikeyan Bhargavan [*] [1], Cédric Fournet [†] [2], Markulf Kohlweiss [‡] [2], Alfredo Pironti [§] [1], Pierre-Yves Strub [¶] [3], and Santiago Zanella-Béguelin [‖] [2]

[1]INRIA
[2]Microsoft Research
[3]IMDEA Software

Draft 0.2, November 2013

Abstract

The TLS Internet Standard features a mixed bag of cryptographic algorithms and constructions, letting clients and servers negotiate their use for each run of the handshake. Although many ciphersuites are now well-understood in isolation, their composition remains problematic, and yet it is critical to obtain practical security guarantees for TLS, as it is implemented and deployed. We experimentally confirm that all mainstream TLS implementations implicitly reuse the same key materials with many different algorithms, some of them of dubious strength. We outline new attacks we found in their handling of session resumption and renegotiation, stressing the need to model multiple related instances of the handshake.

We systematically study the provable security of TLS. To capture the details of the standard and its main extensions, we develop a verified reference implementation of the protocol; we test our implementation against mainstream browsers and servers, for many protocol versions, configurations, and ciphersuites.

We propose new security assumptions for the algorithms used to analyze the handshake, focusing on signatures, key encapsulation, and key derivation. These assumptions are stronger than those expected with simple modern protocols; they enable us to compositionally prove the security of a full-fledged TLS implementation, treated as a detailed 7,000-line model. We present our main definitions, constructions, and proofs for an abstract model of the protocol, featuring series of related runs of the handshake with different ciphersuites. We also describe its refinement to account for the whole reference implementation, based on automated verification tools.

[*]karthikeyan.bhargavan@inria.fr
[†]fournet@microsoft.com
[‡]markulf@microsoft.com
[§]alfredo.pironti@inria.fr
[¶]pierre-yves@strub.nu
[‖]santiago@microsoft.com

Contents

1 Introduction **1**
- 1.1 Prior Security Results on the TLS Handshake . . . 1
- 1.2 Cryptographic Agility in TLS . . . 2
- 1.3 Multiple Sessions and Connections . . . 4
- 1.4 Overview of our Main Results . . . 5

2 Agile Signatures & Certificates **7**

3 Master Secrets & Key Encapsulation **8**
- 3.1 Definition of Agile KEM . . . 8
- 3.2 Carving Agile KEMs out of TLS . . . 9
- 3.3 Tolerating Weak Hash Functions . . . 13

4 Agile PRFs & Key Derivation **14**

5 Defining Agile Security for the TLS Handshake **15**

6 Proving Agile Security for the TLS Handshake **18**

7 Code-Based Verified Implementation **23**
- 7.1 *Control* Interface . . . 23
- 7.2 Message Formats . . . 23
- 7.3 State Machine . . . 24
- 7.4 Performance Evaluation . . . 25

1 Introduction

TLS is the most widely deployed protocol for securing communications and yet, after two decades of attacks, patches, and extensions, its practical security remains controversial. One of the most troublesome aspects of the protocol is its handling of a large number of cryptographic algorithms and constructions. Both the client and the server typically offer many choices, so each run of the handshake involves a negotiation to select the best protocol version, ciphersuite, and extensions available at both ends. This diversity shows no sign of abating, as new extensions are added to the protocol and its implementations, while older features are maintained for backward compatibility. Such a trade-off between functionality and security creates several problems:

(1) It makes the security of TLS depend on its correct configuration, inasmuch as some versions (e.g. SSL2) and algorithms (e.g. MD5 and RC4) are much weaker than others, and may also be subject to different implementation flaws [see e.g. Brumley et al., 2011]. In theory, only very restrictive configurations are provably secure. In practice, dangerous mis-configurations of TLS and its underlying certificates are commonplace [see e.g. Georgiev et al., 2012, Fahl et al., 2013].

(2) It complicates the protocol logic, as the integrity of the negotiation relies itself on the algorithms being negotiated; this is a persistent source of attacks against TLS, from Abadi's 1994 protocol-regression attack till e.g. Langley's 2011 browser fallback attack.

(3) It demands stronger security assumptions on those algorithms, to reflect the fact that honest parties may use the same cryptographic keys with different algorithms. Intuitively, TLS *on its own* enables a range of chosen-protocol attacks, whereby a weak algorithm (chosen by the attacker) may compromise the security of stronger algorithms (chosen by honest parties). We detail below several constructions of TLS that demand joint assumptions on collections of algorithms.

Besides interference between multiple algorithms, TLS features dependencies between multiple runs of the handshake. For instance, a client connection may first run an RSA-based *session* to establish a master secret and a first series of connection keys for the record layer, then run a second session on the same connection, possibly with different algorithms and certificates. In parallel, another client connection may run a *resumption*, relying on the master secret of a prior session to derive new connection keys. At that point, the security of those keys depends on the algorithms and certificates used in three runs of the handshake. This is in sharp contrast with much prior work on the provable security of TLS, which focus on one fixed run of the protocol, for a fixed choice of algorithms.

1.1 Prior Security Results on the TLS Handshake

Cryptographic research on secure key exchange usually follows either a game-based approach or a simulation-based approach, as pioneered by Bellare and Rogaway [1993] and Canetti and Krawczyk [2002], respectively. Gajek et al. [2008] outline a proof of TLS in the simulation-based model of Canetti [2001]. However, Küsters and Tuengerthal [2011] correctly note that their use of a crucial theorem to obtain multi-session security relies on pre-established identifiers not available in TLS, and suggest a framework for overcoming this limitation.

Most of the cryptographic work on TLS follows the game-based approach. Jonsson and Kaliski [2002] analyze the core of the RSA ciphersuites. Morrissey et al. [2008] analyze a variant of the handshake protocol using a modular approach that decomposes the handshake into *pre-master secret*, *master secret*, and *record-key derivation* phases. Both of these works influence our analysis. To pinpoint some differences, Jonsson and Kaliski already propose to use a Key Encapsulation Mechanism (KEM) [Cramer and Shoup, 2003] with one-time nonces, but their KEM includes the record-key derivation and finished messages, and is thus not modular in the sense of Morrissey et al. Although Morrissey et al. show how to boost security using an insecure (only one-way secure) pre-master secret phase, they do not separately model this phase as a KEM. As an advantage of their construction,

they note that the same master secret can be used to derive multiple keys, which turns out to be crucial in our model of resumption. However, they still rely on one-way security for record-key derivation, hence their analysis is more globally dependent on random oracles than ours.

Recently there has been renewed interest in the security (and insecurity) of TLS. Jager et al. [2012] perform a game-based security analysis of the TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA ciphersuite, relying on the analysis of Paterson et al. [2011] for the record protocol. A defining feature of their analysis is that they do not give a definition of TLS handshake security. Instead they define confidential channel establishment (ACCE) security for the whole TLS protocol. Again, this defeats the modularity goals of Morrissey et al. [2008]. Brzuska et al. [2012] propose a more composable game-based analysis technique and use TLS as a case study. They do, however, assume that the key transport encryption scheme is IND-CCA.

Giesen et al. [2012] extend the work of Jager et al. with an analysis of secure session renegotiation, while Krawczyk et al. [2013] extend it to support RSA and server-only authentication ciphersuites. Similarly to Jonsson and Kaliski [2002] and us, they use a KEM abstraction for the cryptographic core of TLS. However, their analysis is still for one fixed ciphersuite at a time, and does not give guarantees when the adversary tricks the client and server into using different algorithms. Moreover, it inherits the monolithic structure of ACCE, which makes it hard to reason modularly about the protocol, e.g. to cover resumption.

The first step in this direction is the work of Bhargavan et al. [2013] on a security proof conducted on a reference implementation of the TLS 1.2 standard, using a combination of type checking and automated verification tools. In the present work, starting from the same code base, we develop a more abstract, human-readable, game-based proof that improves on Bhargavan et al. results and makes them more accessible. In the process, we clarify their definitions and modular structure. In particular we adapt the KEM concept to reason about both the pre-master secret and master secret phases, which allows us to generalize the result of Jonsson and Kaliski [2002], similarly to Krawczyk et al. [2013] but without sacrificing modularity.

1.2 Cryptographic Agility in TLS

Cryptographic *agility* considers the security of constructions and protocols when the same key materials are shared between different algorithms. Acar et al. [2010] propose agile definitions for PRFs and IND-CCA public-key encryption, and advocate cryptographic agility as a major practical concern for TLS.

Some prior works look at the idiosyncratic use of cryptographic primitives in TLS, but do not consider joint security notions; for instance Fouque et al. [2008] and Fischlin et al. [2010] study the use of hash functions in TLS.

We first review the agility mechanisms of TLS, primarily driven by ciphersuites of the form TLS_e_s_WITH_r. Introducing our notations for algorithms, this ciphersuite roughly indicates the KEM e and the (client) signature scheme s for the handshake, and the authenticated encryption scheme r for the record layer. For instance, the commonly-used ciphersuite TLS_RSA_WITH_AES_256_CBC_SHA indicates an RSA handshake with no client certificate, such that the client sends a fresh pre-master secret encrypted under the server public key, and used as the seed of a PRF (using SHA) for deriving 4 keys for SHA-based MACs and AES encryption in CBC mode. TLS 1.2 currently has 314 registered ciphersuites.[1] Still, because of key reuse across algorithms, we stress that the security of TLS does not reduce to the security of a few hundred fixed-algorithm variants of the handshake.

More precisely, the choice of algorithms depends on many other parameters exchanged during the handshake, including the (minimal and maximal) protocol versions proposed by the client, the optional certificate requests and certificate chains, the group description for Diffie-Hellman exchanges, the compression method, and the contents of various extensions in the first two messages of the handshake (e.g. for choosing hash functions and elliptic curves). Working with a reference implementation enables us to achieve this level of precision for our main results.

[1] http://www.iana.org/assignments/tls-parameters/

Empirical study of major web servers and web browsers For 200 out of the top 500 domains,[2] we used an online analyzer[3] to gather extended information on their TLS configuration, including the TLS versions, ciphersuites and extensions they propose. Algorithmic agility seems widely used in practice. On average, the number of proposed ciphersuites is 12, with a standard deviation of approximately 6. In total, 64 ciphersuites are available among the 200 tested websites. Likewise, on average, more than 5 encryption algorithms and 2 hash methods are proposed. Relatively weak algorithms are still widely deployed: 70% of the servers propose at least one ciphersuite based on MD5, and 90% of them propose at least one ciphersuite based on RC4.

All tested websites but one propose at least two TLS versions: 37% of them propose only SSL3 and TLS 1.0; and 56% propose all 4 versions fron SSLv3 till TLS 1.2. Although now forbidden by the standard, 3% of the websites still propose SSL2 with compatible ciphersuites. The (mandatory) secure renegotiation extension is now widely deployed, but 14% of the tested websites are still subject to renegotiation attacks [Ray, 2009]. Concerning session resumption, session tickets are supported by 60% of the servers. All tested servers disable TLS-level compression.

On the client side, we tested 12 implementations, including major web browsers (Chrome, Firefox, Internet Explorer, Safari) and TLS libraries (NSS, OpenSSL, SChannel, Secure Transport). As in the case of servers, the clients analyzed propose a large number of ciphersuites, ranging from 19 to 36, and all of them propose weak hash (MD5) or encryption methods (RC4, or even no encryption). On the other hand, clients tend to support more recent ciphersuites than servers, especially those based on elliptic curves.

Cross-Ciphersuite Attacks in TLS As a first, well-known example of reuse of key materials, most web servers are still configured to use the same RSA certificate both for signing handshake messages and for decrypting pre-master secrets. Accordingly, a game-based reduction to the security of such certificates would involve oracles both for signing and decrypting. Experimentally, 69% of the web servers we tested propose at least one ciphersuite using RSA for encryption and one using it for signing, and we observed that *all* 138 of them use the same key for both purposes. This practice is discouraged and so, for the sake of modularity, our presentation simply treats dual-purpose certificates as compromised; in principle, our results can be extended to cover them under a stronger assumption obtained by merging our definitions for signatures (§2) and KEMs (§3).

As a second example, Mavrogiannopoulos et al. [2012] report an interesting cross-protocol attack between plain Diffie-Hellman (DH) and Elliptic-Curve Diffie-Hellman (ECDH) ciphersuites, due to a mis-interpretation of the signed group description sent by the server. Both families of ciphersuites are (a priori) secure in isolation, but any global configuration enabling a DH client and an ECDH server is subject to their attack.

Our third example concerns the record algorithms (r). Recall that both parties locally derive a key k for r immediately after the KEM phase, and start using those keys before verifying the finished messages to confirm the integrity of the session. (For example, Langley and Moeller [2010] propose as an optimization to start sending encrypted application data before the key confirmation, provided r is trusted.) Depending on r, the key is split into IVs, MAC keys, and encryption keys of various lengths. In particular, the client and the server may securely derive k and then start using it with different algorithms r_C and r_S; for instance the same random bits may be used as an IV in the client and as a MAC key in the server. To our knowledge, we are the first to report this cross-algorithm attack. We have not built an exploit based on two TLS record algorithms (r_C, r_S), but we can easily design a pair of schemes strong in isolation and subject to this attack, and we suspect that e.g. key recovery attacks against RC4 in r_C can be used to attack r_S. Remarkably, as explained below, our model fully accounts for this agility feature of TLS 1.2, *without the need for joint security assumptions* on the constructions r used in the record layer.

[2] http://www.alexa.com/topsites/global excluding domains that do not support HTTPS with a valid certificate.
[3] https://www.ssllabs.com/ssltest/analyze.html

1.3 Multiple Sessions and Connections

We set up some TLS terminology for multiple related handshakes. Local instances of the protocol provide a *connection* (concretely, taking ownership of a TCP connection), either as client or server. Each connection goes through a sequence of *epochs*, each epoch running one *handshake*. We refer to the epochs performing full handshakes as *sessions*, and to the epochs performing abbreviated handshakes as *resumptions*. We have a transition from the current epoch to the next as the local instance *completes*, by successfully processing the last message of the handshake. Abstractly, the local instance never stops; it is then ready to send (or receive) the first message of the next handshake.

Sessions (intend to) establish a fresh *master secret*, associated with a series of variables extracted from the handshake messages that record its origin and purpose, such as a unique label (obtained by concatenating client and server random values), an algorithm description a (primarily the protocol version and the negotiated ciphersuite) and optional certificate chains for the client and the server. The master secret is immediately used to derive fresh keys for the record layer. *Resumptions* instead rely on an existing complete session to save the cost of public-key cryptography and directly derive fresh keys for the record layer, using the algorithms and the master secret of the original session with a fresh unique label (obtained by concatenating the new client and server random values).

For each epoch, the handshake consists of a series of messages exchanged using the current record-layer protection mechanisms, initially in plain text, then typically using authenticated encryption. A feature of TLS that traditionally resists abstraction is that the handshake delivers the newly-established algorithms and keys for the current epoch to the record layer *before* the epoch completes, so that the last messages of the handshake can be exchanged within records protected by the new keys. In this work, we primarily model the handshake, not TLS as a whole. Hence, following the standard, our main definition returns the derived record keys in the middle of the handshake, before signalling its completion a few messages later. Since the handshake does *not* rely on record-layer protection, we can safely ignore its details, by assuming that the handshake adversary controls both the network and the record layer. (Our reference implementation also includes verified code for the record layer, showing the usability of the keys established by the handshake; its security model ensures by typing that the record keys are used for protecting application data only after completion. This resolves the *finished message controversy* of Jager et al. [2012] in a novel and surprisingly elegant way.)

Before describing our results in more detail, we present two attacks that motivate the need to precisely model agility, completions, and resumptions.

Sending encrypted data optimistically (Sample Attack) In recent years, many proposals have tried to improve the latency of TLS connections by reducing the number of roundtrips that an application needs to wait for in the TLS handshake before sending data. The False Start extension (implemented by all Google servers and various browsers) allows the client to send data before the handshake is complete. Independent proposals try to improve the privacy of the handshake parameters by sending some messages encrypted, before the handshake is complete. The Next Protocol Negotiation (NPN) extension (implemented by all major websites and browsers) sends an encrypted message after the cipher state changes but before the handshake is complete. In both cases, it is easy to get the implementation wrong so that confidential data is leaked to the adversary:

• If an encrypted message is sent before the server certificate has been verified (or authorized by the user), then the message may be sent to the adversary. We found such attacks on both Chrome and Firefox where a network attacker was able to recover privacy-sensitive data even though both client and server were honest.

• If an encrypted message is sent after certificate verification but before the handshake is complete, then its security relies on the ciphersuites accepted by the client, since a man-in-the-middle adversary will be able to downgrade the client to its weakest ciphersuite regardless of the server (in a variation of protocol regression). In response, extensions like False Start restrict the agility of the TLS handshake by requiring the ciphersuite to use symmetric ciphers with keys larger than 128 bits (RC4, AES) and strong key-exchange methods (DHE_RSA, ECDHE_RSA, DHE_DSS, ECDHE_ECDSA). However, for example, MD5 is still allowed as a hash algorithm

during False Start.

Storing and resuming sessions with the adversary (Sample Attack) When the initial TLS handshake completes, it establishes both a connection (secure channel) and a resumable, long-term session. Key materials established in this handshake, notably the master secret, are stored at both client and server in a session database. So, if the client tries to connect to the server again, it will first look at the database to check if it already has a live session, and if so will reuse the session key materials to establish a new connection through an abbreviated handshake, hence reducing the latency of connection establishment. Notably entity authentication is not repeated during the abbreviated handshake; the new connection is presumed to have the same client and server principals. However, storing and indexing this session database is left to the implementation, leading to a number of vulnerabilities and threats:

- If an adversary is able to obtain the master secret of a live session, it can impersonate both the client and the server to each other.
- If the client uses the IP address of the server as an index into the database, instead of its public key (or certificate DNS name), then an active network attacker can use DNS poisoning to fool the client into thinking that a session between the client and an attacker is in fact between the client and the server, and hence obtain sensitive information that the client was going to send to the server.
- If the client stores sessions into its database before the server certificate is verified, then an active network attacker can again fool the client into resuming a session with the honest server using a session controlled by the attacker. If the initial handshake used an RSA key exchange, the abbreviated handshake will succeed, with the honest client and server connected to each other using keys that are known to the attacker. We found such an attack on Chrome.

The underlying tension leading to these kinds of vulnerabilities is the need to reduce latency while keeping (or strengthening) security guarantees. The attacks mentioned above (on Chrome and Firefox) appear in other TLS-based software in both stronger and weaker forms. We are in the process of responsibly disclosing these attacks. We will be able to report on possible fixes after disclosure.

1.4 Overview of our Main Results

Our main result is the provable security of a reference implementation of the TLS handshake, seen as a very detailed, standard-compliant cryptographic model of the protocol. Overall, our code for the handshake consists of 4,000 lines of ML; it is part of a full-fledged 7,000 line implementation of TLS.

We see the use of a verified reference implementation as essential to capture the amount of details necessary to account for multiple related algorithms and multiple related epochs in the TLS handshake. §7 describes our use of high-level programming, type systems, and automated provers to carry out a modular cryptographic proof at this scale. It also provides experimental results, showing that our "model" suffices to run sample client and server applications and to interoperate with mainstream implementations of TLS with comparable performance.

To present this result and explain its proof structure, however, we rely on more succinct definitions and constructions, given in §2–6. This more abstract treatment suffices to convey the main ideas, but it elides many important aspects of the handshake—we refer to the standard Dierks and Rescorla [2008] or the implementation. For example, we deliberately avoid any specification of the message formats, or the procedure for selecting algorithms from their contents.

We verify TLS security relative to joint security assumptions for different combinations of algorithms.

Agile Signatures and Certificates We begin with a reasonably simple definition for agile, public-key signatures. TLS combines three core signatures algorithms $s \in \{\mathsf{RSA}, \mathsf{DSA}, \mathsf{ECDSA}\}$, each used with a range of hash algorithms to compress the text before signing. Embarrassingly, neither TLS nor X.509 certificates (as used for

TLS) support an explicit policy about which combinations are enabled, so we need a notion of security that can withstand the presence of *some* bad hash algorithms in the standard, such that, for example, a verifier tricked into using MD5 remains secure as long as the signer does not use MD5, and vice-versa.

Thus, for each signature algorithm s, we define security against h-H existential forgeries, as security against an adversary that must forge a valid signature for algorithms (s, h), given access to a signing oracle for any algorithms (s, h') where $h' \in H$. We show that the combination of secure schemes may not be jointly secure. We give a simple construction, but leave open the concrete analysis of the range of algorithms supported by TLS.

Our presentation also avoids a specific model for certificate chains and their management. Our handshake authenticates the exchanged certificate chains, and relies on the last signature of each chain (on a key exchange value, or on a series of handshake messages), but otherwise treats those chains as bitstrings to be verified and authorized by the TLS application once a session completes.

Key Encapsulation Our agile definition for KEM abstracts over the two main families of core constructions for key exchange: RSA and Diffie-Hellman ephemeral (DHE) and static (DH) exchanges, based on plain or elliptic-curve groups. We let e range over these constructions. (Our implementation does not currently support ECDH, and would require extra checks on group descriptors to prevent the attack of Mavrogiannopoulos et al. 2012.) In TLS, each of these constructions is also parameterized by a hash algorithm, applied to the pre-master secret to obtain a (pseudo-) random master secret, so we arrive at a security definition parameterized by the algorithm used by the decryptor and the set of algorithms used by the encryptor.

Many troubling attacks on the TLS KEMs have been reported. See for example Bleichenbacher [1998] attack on RSA PKCS#1, significantly improved by Bardou et al. [2012], and its specialization to TLS by [Klima et al., 2003].

Key Derivation and Finished Messages The master secret established in a TLS session is used repeatedly in each epoch relying on that session, for two main purposes:

(1) deriving the authenticated-encryption key materials k for the epoch; and

(2) computing the MAC of the log of all fragments exchanged in the epoch to verify its integrity.

(Anecdotically, in SSL3 only, it is also used to MAC the log before signing in `CertificateVerify` messages, and it may be used with different algorithms for those purposes.)

We model key derivation in two steps, first as an h-indexed agile family of PRFs, then as an agile functionality that separates its different usages and ensures that the derived key is used with the same algorithms by the client and by the server. (The PRF assumption is convenient for both usages, but a weaker EUF-CMA-like assumption would suffice for integrity.)

An Abstract Model for Sequences of Handshakes The main two purposes of the handshake are to establish shared symmetric keys for protecting message fragments, and to agree on many parameters, notably those used in the handshake itself. We refer to Figure 1 and the appendixes for additional details.

Our main definition is parameterized by a predicate α on the choice of all algorithms a in a given epoch, such that we obtain security guarantees only when $\alpha(a)$ holds. Our main theorem reduces the concrete security of the handshake to a series of agile assumptions on its underlying constructions for signing, for KEM, and for PRFs.

Summary of Contributions We provide (1) Experimental data on algorithm agility in typical web client and web server deployments.

(2) New attacks.

(3) A reference implementation of the handshake, supporting the common features of the standard and its main extensions, tested against mainstream TLS clients and servers using 4 versions ranging from SSL3 to TLS1.2, 12 ciphersuites, and various certificates and extensions.

(4) New, agile security definitions for the core algorithms used by TLS, parameterized by (at least) a choice of hash algorithms: for signatures; for master secret key encapsulation (capturing both RSA-based and Diffie-Hellman-based ciphersuites), for the finished-message MACs, and for agile key derivation for the record layer.

(5) A new, parametric definition of handshakes, more abstract, but compatible with our reference implemen-

tation.

(6) A modular, provably secure construction for a (relatively) abstract TLS handshake.

(7) A proof that our implementation meets this definition, based on the same modular structure, but at a greater level of detail, relying on automated type-based verification for scalability.

To our knowledge, we provide the first provable-security results for TLS that account for algorithm agility. We are also the first to fully model the security of handshakes related by (session) resumption and (connection) renegotiation. Our work sheds light on important design and implementation issues of TLS as it is used today, notably cryptographic agility and interdependent handshakes; it suggests simple improvements to strengthen its security.

Notation. We use the following notation for assignment and random sampling: $:=$ denotes deterministic assignment, $\leftarrow$ a random assignment either uniformly from a set S or according to a distribution determined by an algorithm Alg and its random choices. We denote by $\$_{ms}$ and $\$_r$ the set of master secrets and record keys for algorithms r respectively. When describing generic key exchange or key derivation primitives we use $\$$ to denote the key space.

We use a special font for algorithms, e.g., Alg. If such an algorithm uses a more primitive algorithm in its construction, we denote it by alg. If in the security definition the adversary is given control over Alg via an oracle, we denote the oracle by ALG.

2 Agile Signatures & Certificates

TLS employs hash-then-sign signature schemes for which the hash algorithm can vary depending on the protocol version, ciphersuite and extensions. This is an agility problem, as the same signing key is used with different signature schemes. Hence, we verify TLS security relative to joint security assumptions for different combinations of hash algorithms. We let s and h range over signature and hash algorithm names, respectively.

An *agile signature scheme* consists of algorithms $\mathsf{KeyGen}_s()$, $\mathsf{Sign}_s(h, sk, msg)$, $\mathsf{Verify}_s(h, pk, msg, \sigma)$. Depending on the agility parameter h the signing key sk is used with a different concrete signature scheme. For TLS, the *agile (hash-then-sign) signature scheme* with base algorithm $(\mathsf{keygen}_s, \mathsf{sign}_s, \mathsf{verify}_s)$ and hash algorithm H_h consists of three algorithms:

- $\mathsf{KeyGen}_s()$ generates a key pair for algorithm s using $\mathsf{keygen}_s()$;
- $\mathsf{Sign}_s(h, sk, msg)$ signs a message using algorithms h and s as $\mathsf{sign}_s(sk, \mathsf{H}_h(msg))$;
- $\mathsf{Verify}_s(h, pk, msg, \sigma)$ verifies a message with tag σ using algorithms h and s as $\mathsf{verify}_s(pk, \mathsf{H}_h(msg), \sigma)$.

We define existential unforgeability under chosen message attacks (EUF-CMA) for agile signatures.

Definition 1 (EUF-CMA) *Consider the game below, parameterized by H for an adversary $\mathbf{A}$ with access to a signing oracle* SIGN.

$pk, sk \leftarrow \mathsf{KeyGen}_s()$	$\underline{\mathsf{SIGN}(h', msg)}$
$Q := \{\}$	$\mathbf{if}\ h' \notin H\ \mathbf{then\ return}\ \perp$
$msg^\star, \sigma^\star \leftarrow \mathsf{A}^{\mathsf{SIGN}(\cdot,\cdot)}(pk)$	*add msg to Q*
	$\mathbf{return}\ \mathsf{Sign}_s(h', sk, msg)$

The scheme $(\mathsf{KeyGen}_s, \mathsf{Sign}_s, \mathsf{Verify}_s)$ is (ϵ, t, h, H)-secure when, for all adversaries running in time t, the probability that $msg^\star \notin Q$ and $\mathsf{Verify}_s(h, pk, msg^\star, \sigma^\star) = 1$ is at most ϵ.

Our definition generalizes standard, non-agile EUF-CMA security, which coincides with $(\epsilon, t, h, \{h\})$-security for a scheme with fixed hash algorithm h. For example, assume given a base EUF-CMA scheme using algorithms

(s, h) and let $\mathsf{H}_{h'}(m) = \mathsf{H}_h(\pi(m))$ for some efficiently invertable permutation π. As an agile scheme, this scheme is both $(\epsilon, t, h, \{h\})$ and $(\epsilon, t, h', \{h'\})$-secure. However, an adversary can return m, $\mathsf{SIGN}(h', \pi^{-1}(m))$ for any m such that $m \neq \pi^{-1}(m)$ to trivially break $(\epsilon, t, h, \{h, h'\})$-security.

On the other hand, if H_h is collision resistant, H_h and $\mathsf{H}_{h'}$ have disjoint range, and the base signature scheme $(\mathsf{keygen}_s, \mathsf{sign}_s, \mathsf{verify}_s)$ itself is (ϵ, t)-EUF-CMA secure on the range of H_h and $\mathsf{H}_{h'}$; then we have $(\epsilon', t', h, \{h, h'\})$-security for the agile hash-then-sign signature scheme (where the difference between ϵ, t and ϵ', t' depend on the reduction to collision resistance).

Note that the base signature schemes used in TLS are not EUF-CMA secure. The best we can do is thus to assume that the hash-then-sign signature scheme meets Definition 1. Another feature of TLS is that it does not authenticate the hash algorithm used to compress the message, in particular, under our definition H could in principle contain the same hash algorithm twice, under different names.

3 Master Secrets & Key Encapsulation

In principle, TLS key encapsulation is parameterized by a triple of algorithm descriptors (e, h, H), where e is the first part of the ciphersuite (minus the signature algorithm), h is the hash used by the client, and H is the set of hashes used by the server. (We will also use h, H to encode some TLS oddities, like tweaks on the KEM scheme that depend on the protocol version. In these cases we refer to the actual hash algorithm and admissible hash algorithm set by $\bar{h}$ and $\bar{H}$ respectively; intuitively $\bar{\cdot}$ turns a tuple containing a hash algorithm name or a set of such tuples into said name or set of names respectively.) Thankfully, TLS does not mix RSA and DH, so e can be inlined in the KEM instance.

An *agile key encapsulation scheme* (agile KEM) based on a public key scheme e consists of three algorithms:

- $\mathsf{KeyGen}_e()$ generates a keypair for algorithm e.
- $\mathsf{Enc}_e(h, pk, \ell)$ generates a ciphertext c and a key k using algorithms e and h, and nonce ℓ.
- $\mathsf{Dec}_e(h, sk, \ell, c)$ computes a key (or $\perp$) from c and sk using algorithms e and h, and nonce ℓ.

In our security definition below, we capture our assumptions about the use of nonces for master secret generation, namely that at most one KEM ciphertext is decrypted by honest servers using the same nonce.

3.1 Definition of Agile KEM

We define indistinguishability for an agile tagged KEM under replayable chosen ciphertext attacks with nonces (IND-RCCA-N). This is a relaxation of CCA security and was first considered for public key encryption by Canetti et al. [2003].

Definition 2 (KEM) *Consider the game below, parameterized by h and H, s.t. $h \in H$, using two sets of labels Q_ℓ and keys Q_k, for an adversary $\mathbf{A}$ given oracle access to* ENC *and* DEC*:*

$pk, sk \leftarrow \mathsf{KeyGen}_e()$
$b \leftarrow \{0, 1\}$
$Q_\ell := \{\}$
$Q_k := \{\}$
$b' \leftarrow \mathbf{A}^{\mathsf{ENC}(\cdot,\cdot),\mathsf{DEC}(\cdot,\cdot,\cdot)}(pk)$

$\underline{\mathsf{ENC}(\ell)}$
$c, k_0 \leftarrow \mathsf{Enc}_e(h, pk, \ell)$
add k_0 and k_1 to Q_k
$k_1 \leftarrow \$$
return c, k_b

$\underline{\mathsf{DEC}(h', \ell, c)}$
if $\ell \in Q_\ell \vee h' \notin H$ **then return** $\perp$
add ℓ to Q_ℓ
$k \leftarrow \mathsf{Dec}_e(h', sk, \ell, c)$
if $k \in Q_k$ **then return** $\perp$
return k

The scheme $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is (ϵ, t, h, H)-IND-RCCA-N when, for any adversary $\mathbf{A}$ running in at most time t, the probability that $b = b'$ is bounded by $\frac{1+\epsilon}{2}$.

This game can be simplified to allow only for a single call to the challenge oracle ENC. This is shown by a simple hybrid argument that returns k_1 for all challenges up to i and k_0 for all challenges greater than i. If an attacker $\mathbf{A}$ can distinguish between Hybrid 0 and Hybrid $|Q_k|$,[4] then he must also be able to distinguish between Hybrid i and Hybrid $i-1$ for some i. We can thus build reductions $\mathbf{B}_i$ from $\mathbf{A}$ that break single challenge IND-RCCA-N. From now on, when proving IND-RCCA-N security we will primarily consider this single challenge case. When reasoning about the security of TLS, the multi-challenge version is more convenient and simplifies our theorem statement.

Theorem 1 (Single challenge to multi challenge) The scheme $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is (ϵ, t, h, H)-IND-RCCA-N, if it is (ϵ', t', h, H)-IND-RCCA-N for adversaries that query ENC only once, where $\epsilon \leq |Q_k|\epsilon'$ and t' is t plus the cost of simulating the hybrids.

3.2 Carving Agile KEMs out of TLS

We take the same starting point as Krawczyk et al. [2013] and model the basic key-exchange functionality of TLS as different variations on KEMs. This is a useful abstraction, as it allows us to model both RSA and DH using the same formalism. We prove statements about both ephemeral DH and (semi)[5] static DH.

We are, however, more modular and consider pre-master secret, master secret, and record key generation as separate phases in the protocol. The basic KEM mechanisms $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$, $e \in \{RSA, DH\}$ secret (short *pms*-KEM) are *not* secure KEMs under any indistinguishability notion even under relatively weak active attacks, such as plaintext checking attacks (PCA): recall the Bleichenbacher attack, and the lack of active security for basic Diffie-Hellman (e.g., querying the plaintext-checking oracle on c^r and pms^r, for any $r \neq 1$, suffices to win the indistinguishability game).

***pms*-KEM for RSA** Algorithm keygen_{RSA} generates a fresh RSA key pair pk, sk; $\mathsf{enc}_{RSA}(pv, pk)$ generates an RSA pre-master secret *pms* that encodes the protocol version pv and encrypts the pre-master secret using PKCS#1v1.5 to obtain a ciphertext c; $\mathsf{dec}_{RSA}(pv, sk, c)$ decrypts the PKCS#1v1.5 ciphertext; if decryption fails, it returns a random *pms* with the pv passed as the argument.

***pms*-KEM for DH** For all Diffie-Hellman ciphersuites, keygen_{DH} selects the group parameters pp, generates public and private DH values (g^x, x), and returns $pk = (pp, g^x)$ as the public and $sk = (pk, x)$ as the private KEM keys; $\mathsf{enc}_{DH}(pv, pk)$ samples y and returns $c = g^y$ and $pms = g^{xy}$; $\mathsf{dec}_{DH}(pv, (pk, x), c)$ checks that c is in the group and returns $c^x = g^{xy}$.

We once more stress that, on their own, the *pms*-KEMs of TLS provide no meaningful security guarantees. Rather than using *pms* as a key, TLS instead feeds it through a key extraction function (KEF), parameterized by hash functions, to extract the master secret *ms*. This can be seen as a *encapsulate-then-hash* approach for KEMs in analogy to the *hash-then-sign* approach for signatures.

The resulting *ms*-KEM is conjectured to provide adequate security for TLS, though to our knowledge this has never been formally stated or proven. The work of Jonsson and Kaliski [2002] and Krawczyk et al. [2013] instead view an even larger part of TLS as a KEM that directly derives the record key material for authenticated encryption, including the finished messages. Krawczyk et al. prove that such a KEM, which we call a *record*-KEM, is constrained chosen ciphertext attack (IND-CCCA) secure [Hofheinz and Kiltz, 2007]—the constraint is that the decrypted client finished message verifies. Instead we show that the intermediary *ms*-KEM for extracting the master secret is IND-RCCA-N secure. Our approach is more modular, enabling us to better capture session resumption and certain TLS extensions like False Start that rely on materials derived from the master secret before verifying the finished message. The downside is that we have to make an additional assumption in the proof.

[4] We abuse notation and use $|Q_k|$ to refer to the maximum number of calls to ENC.

[5] Like Krawczyk et al., we require that clients are 'ephemeral' and compute a fresh KEM ciphertext for each handshake.

Generic *ms*-KEM construction We now describe how TLS constructs an *ms*-KEM from a *pms*-KEM $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ and a key extraction function family $\{\mathsf{KEF}_{\bar{h}}\}_{\bar{h}}$ indexed by hash algorithm names $\bar{h}$:

- $\mathsf{KeyGen}_e() \triangleq \mathsf{keygen}_e()$.
- $\mathsf{Enc}_e((pv, \bar{h}), pk, \ell) \triangleq c, pms \leftarrow \mathsf{enc}_e(pv, pk); ms \leftarrow \mathsf{KEF}_{\bar{h}}(pms, \ell); \mathbf{return}\ c, ms$.
- $\mathsf{Dec}_e((pv, \bar{h}), sk, \ell, c) \triangleq pms \leftarrow \mathsf{dec}_e(pv, sk, c); ms \leftarrow \mathsf{KEF}_{\bar{h}}(pms, \ell); \mathbf{return}\ ms$.

Our definition makes explicit that in TLS, h encodes further configuration options besides the hash function algorithm used for master secret extraction, that is, $h = (pv, \bar{h})$ where *pv* is the protocol version.

We now give some preliminary theorems and conjectures about agile IND-RCCA-N KEMs to relate them to prior work and more standard cryptographic assumptions. We hope that these definitions and assumptions will stimulate further cryptanalytic work on TLS. As explained above, we consider the single challenge case. We first define the assumptions used by our main theorem: *non-randomizability under plaintext checking attacks* (NR-PCA) and *one-wayness under plaintext checking attacks* (OW-PCA).

Definition 3 (NR-PCA, OW-PCA) *Consider the games below for adversaries* $\mathbf{A}$ *with oracle access to* PCO*:*

Non-Randomizability Game	*One Way Game*	$\mathsf{PCO}(pv, pms, c)$
$pk, sk \leftarrow \mathsf{keygen}_e()$	$pk, sk \leftarrow \mathsf{keygen}_e()$	$pms' \leftarrow \mathsf{dec}_e(pv, sk, c)$
$pv, state \leftarrow \mathbf{A}(pk)$	$pv, state \leftarrow \mathbf{A}(pk)$	$\mathbf{return}\ pms = pms'$
$c, pms \leftarrow \mathsf{enc}_e(pv, pk)$	$c, pms \leftarrow \mathsf{enc}_e(pv, pk)$	
$pv^*, c^* \leftarrow \mathbf{A}^{\mathsf{PCO}(\cdot,\cdot,\cdot)}(state, c)$	$pms^* \leftarrow \mathbf{A}^{\mathsf{PCO}(\cdot,\cdot,\cdot)}(state, c)$	
$\mathbf{return}\ c \neq c^* \wedge pms = \mathsf{dec}_e(pv^*, sk, c^*)$	$\mathbf{return}\ pms = pms'$	

The scheme $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ *is* (ϵ, t)*-NR-PCA (resp.* (ϵ, t)*-OW-PCA) when, for any adversary* $\mathbf{A}$ *running in at most time* t*, the probability that the first (resp. second) game returns 1 is bounded by* ϵ*.*

Conjecture 1 (Informal: Non-randomizability of RSA *pms*-KEM) *It seems hard to randomize an RSA pms-KEM ciphertext into a different ciphertext that results in the same pms.*

NR-PCA seems to be implied by the common-input extractability *assumption of Barthe et al. [2012] and OW-PCA, once one inverses the role of randomness and plaintexts and takes the oddities of PKCS#1v1.5 encoding and its use in TLS into consideration.*

Note that the DH *pms*-KEM is trivially non-randomizable, as it has unique ciphertexts.

Conjecture 2 (Informal: OW-PCA security of RSA *pms*-KEM) *Jonsson and Kaliski [2002] gives us reason to conjecture that* $(\mathsf{keygen}_{RSA}, \mathsf{enc}_{RSA}, \mathsf{dec}_{RSA})$ *is* (ϵ, t)*-OW-PCA secure under the* (ϵ', t')*- partial-RSA decision oracle assisted RSA inversion assumption where* ϵ', t' *are however not tight.*

The following theorem considers only a single query to the challenge oracle ENC and we rely on Theorem 1 to obtain multi-challenge security.

Theorem 2 (IND-RCCA-N from NR-PCA and OW-PCA KEM in the ROM) The *ms*-KEM $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is single challenge (ϵ, t, h, H)-IND-RCCA-N secure for $\epsilon \leq 2\epsilon_R + \epsilon_O$ if

(1) we heuristically assume that all $\mathsf{KEF}_{\bar{h}}$, $\bar{h} \in \bar{H}$ with $\bar{H} = \{\bar{h} \mid (pv, \bar{h}) \in H\}$ behave like independent random oracles,

(2) the underlying *pms*-KEM $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ is (ϵ_R, t_R)-NR-PCA, where t_R is t plus the cost of running the IND-RCCA-N game,

(3) the underlying *pms*-KEM $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ is (ϵ_O, t_O)-OW-PCA secure, where t_O is t plus the cost of running the reduction $\mathbf{B}$ (see proof below).

As we model the *ms*-KEM in the random oracle model (ROM), we in fact prove security under the following definition which models multiple internal key extraction functions $\mathsf{KEM}_{\bar{\mathsf{h}}}$, $\bar{h} \in \bar{H}$, as random oracles.

Definition 4 (IND-RCCA-N in multi KEF ROM) *Consider the game below, parameterized by h and H, using two sets for labels Q_ℓ and keys Q_k and an associative map Q, for an adversary $\mathbf{A}$ given oracle access to* ENC*,* DEC*, and* KEF*:*

$pk, sk \leftarrow \mathsf{KeyGen}_e()$
$b \leftarrow \{0,1\}$
$Q := \{\}$
$Q_\ell := \{\}$
$Q_k := \{\}$
$b' \leftarrow \mathbf{A}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot,\cdot,\cdot),\mathsf{KEF}(\cdot,\cdot,\cdot)}(pk)$

$\underline{\mathsf{ENC}(\ell)}$
$c, k_0 \leftarrow \mathsf{Enc}_e(h, pk, \ell)$
add k_0 and k_1 to Q_k
$k_1 \leftarrow \$$
return c, k_b

$\underline{\mathsf{DEC}(h', \ell, c')}$
if $\ell \in Q_\ell \vee h' \notin H$ **then return** $\perp$
add ℓ to Q_ℓ
$k \leftarrow \mathsf{Dec}_e(h', sk, \ell, c')$
if $k \in Q_k$ **then return** $\perp$
return k

$\underline{\mathsf{KEF}(\bar{h}', pms, \ell)}$
if $\bar{h}' \notin \bar{H}$ **then return** $\perp$
if $Q(\bar{h}', pms, \ell) = \perp$ **then** $Q(\bar{h}', pms, \ell) \leftarrow \$_{\bar{h}'}$
return $Q(\bar{h}', pms, \ell)$

The scheme $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is (ϵ, t, h, H)-IND-RCCA-N in the random oracle model when, for any adversary $\mathbf{A}$ running in at most time t, the probability that $b = b'$ is bounded by $\frac{1+\epsilon}{2}$.

Recall that Enc_e and Dec_e make internal use of the $\mathsf{KEF}_{\bar{h}}$ to derive keys from pre master secrets *pms*. Proofs in the random oracle model only provide heuristic guarantees. As weaker hash algorithms like MD5 are still widely supported by TLS, this is particularly problematic for TLS as it is used today. We investigate ways to avoid the random oracle assumption for hash algorithms in $H \setminus h$ below, but it is instructive to consider the setting where all KEF functions are modeled as random oracles first.

PROOF: The proof proceeds using a sequence of games. Let $\mathsf{Pr}^{Gi}_{\Pi}(b' = 1)$ be the probability that $\mathbf{A}$ outputs 1 in Game i.

- *Game 1.* This is the IND-RCCA-N game for $b = 0$.
- *Game 2.* The same as Game 1, except that we abort if $\mathbf{A}$ queries DEC using ℓ and $c' \neq c$ such that ℓ was queried to ENC and c' internally decrypts to the challenge *pms*. (Note that this would lead to Dec returning k_0, something that cannot be detected by a reduction that is only given c.)

  If $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ is (ϵ_R, t)-NR-PCA, then $|\mathsf{Pr}^{G1}_{\Pi}(b' = 1) - \mathsf{Pr}^{G2}_{\Pi}(b' = 1)| \leq \epsilon_R$.

  The reduction $\mathbf{B}_1$ receives the NR-PCA public key and has access to the PCA oracle. It maintains a table to simulate random oracle queries for all $h \in H$. A record $(h, \mathsf{KEF}, pms, \ell, k)$ means that $\mathbf{B}_1$ returned k on a direct query by the adversary to $\mathsf{KEF}(\bar{h}, pms, \ell)$. A record $(h, \mathsf{DEC}, c, \ell, k)$ means that $\mathbf{B}_1$ returned k on a query by the adversary to $\mathsf{DEC}(h, \ell, c)$. Repeated queries to KEF are answered using the table, while queries to DEC cannot be repeated because of the nonce restrictions.

  We use the PCA oracle to keep these two types of records consistent: On a query $\mathsf{KEF}(\bar{h}, pms, \ell)$ we check whether there is a record $(h, \mathsf{DEC}, c, \ell, k)$ such that $\mathsf{PCA}(pv, pms, c) = 1$. If such a record is found we return k otherwise we sample a random k. In both cases we update the table and add a KEF record. On a query $\mathsf{DEC}(h, \ell, c)$ we check whether there is a record $(h, \mathsf{KEF}, pms, \ell, k)$ such that $\mathsf{PCA}(pv, pms, c) = 1$. If such a record is found we return k otherwise we sample a random k. In both cases we update the table and add a DEC record.

  The reduction outputs the ciphertext c that is sent to DEC with the same ℓ as used with the ENC oracle query. $\mathbf{B}_1$ wins the game whenever Game 1 and 2 would result in different views for $\mathbf{A}$.

- *Game 3*. The same as Game 2, except that ENC now behaves as if $b = 1$.

  If $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ is (ϵ_O, t)-OW-PCA then $|\mathsf{Pr}^{G2}_{\Pi}(b' = 1) - \mathsf{Pr}^{G3}_{\Pi}(b' = 1)| \leq \epsilon_O$.

  The reduction $\mathbf{B}_2$ receives the OW-PCA public key and has access to the PCA oracle. It maintains a table to simulate random oracle queries for all $h \in H$. A record $(h, \mathsf{KEF}, \mathit{pms}, \ell, k)$ means that $\mathbf{B}_2$ returned k on a direct query by the adversary to $\mathsf{KEF}(\bar{h}, \mathit{pms}, \ell)$. A record $(h, \mathsf{DEC}, c, \ell, k)$ means that $\mathbf{B}_2$ returned k on a query by the adversary to $\mathsf{DEC}(h, \ell, c)$. For both types of queries, repeated queries are answered using the table.

  We use the PCA oracle to keep these two types of records consistent: On a query $\mathsf{KEF}(\bar{h}, \mathit{pms}, \ell)$ we check whether there is a record $(h, \mathsf{DEC}, c, \ell, k)$ such that $\mathsf{PCA}(\mathit{pv}, \mathit{pms}, c) = 1$. If such a record is found we return k otherwise we sample a random k. In both cases we update the table and add a KEF record. On a query $\mathsf{DEC}(h, \ell, c)$ we check whether there is a record $(h, \mathsf{KEF}, \mathit{pms}, \ell, k)$ such that $\mathsf{PCA}(\mathit{pv}, \mathit{pms}, c) = 1$. If such a record is found we return k otherwise we sample a random k. In both cases we update the table and add a DEC record.

  In the random oracle model, the only difference between Game 2 and Game 3, is that for Game 2 $\mathsf{KEF}_{\bar{h}}$, modeled as a random oracle, is internally queried on the *pms* generated by the ENC oracle. $\mathbf{B}_2$ thus embeds the OW-PCA ciphertext as the c returned by the ENC oracle together with a random k. Any successful $\mathbf{A}$ must query KEF on the plaintext *pms* of c with probability at least ϵ, in which case $\mathbf{B}_2$ can detect using $\mathsf{PCA}(\mathit{pv}, \mathit{pms}, c)$ that it wins the game by outputting *pms*.

- *Game 4*. The same as Game 3, except that DEC now again accepts ciphertexts c that decrypt to the challenge *pms*. Reduction $\mathbf{B}_3$ is similar to $\mathbf{B}_1$.

  Game 4 behaves just like the IND-RCCA-N game for $b = 1$. Thus $\epsilon \leq 2\epsilon_R + \epsilon_O$.

Theorem 3 (OW-PCA security of DH *pms*-KEM) $(\mathsf{keygen}_{DH}, \mathsf{enc}_{DH}, \mathsf{dec}_{DH})$ is (ϵ, t)-OW-PCA secure under the (ϵ, t')-GapCDH assumption [Okamoto and Pointcheval, 2001], where t' is essentially t.

PROOF: The reduction $\mathbf{B}$ receives pp, g^x, g^y as input and has access to a DDH oracle $\mathsf{DDH}(X, Y, Z)$. It returns (pp, g^x) as *pk*. When queried on ENC, it returns g^y as c. Plaintext checking queries $\mathsf{PCA}(\mathit{pv}, c, \mathit{pms})$ are answered by $\mathsf{DDH}(g^x, c, \mathit{pms})$. When $\mathbf{A}$ succeeds in computing $\mathit{pms} = g^{xy}$, $\mathbf{B}$ forwards this value to win the GapCDH game.

Theorem 4 (PRF-ODH a special case) The *ms*-KEM $(\mathsf{KeyGen}_{DH}, \mathsf{Enc}_{DH}, \mathsf{Dec}_{DH})$ is $(\epsilon, t, h, \{h\})$-IND-RCCA-N under the (ϵ, t)-PRF-ODH assumption for the group parameters *pp* generated by KeyGen_{DH} and the pseudo-random function $f_{g^{uv}}(\cdot)$ defined as $\mathsf{KEF}(\bar{h}, g^{uv}, \cdot)$.

In fact under the PRF-ODH formulation of Krawczyk et al. [2013] $(\mathsf{KeyGen}_{DH}, \mathsf{Enc}_{DH}, \mathsf{Dec}_{DH})$ is $(\epsilon, \mathcal{A}, h, \{h\})$-IND-CCA secure, even if TLS would allow the reuse of nonces.

Conjecture 3 (Informal: IND-RCCA-N security of TLS *ms*-KEMs) *Under the GapCDH assumption, the partial-RSA decision oracle assisted RSA inversion assumption and the common-input extractability assumption, the ms-KEMs used in TLS are IND-RCCA-N secure in the random oracle model.*

Conjecture 4 (Informal: Relation to Bhargavan et al. [2013]) *Under the non-standard assumptions made by Bhargavan et al. about the* KEF *(which they call a CRE for computational randomness extractor) we have the following two conjectures:*

(1) *If the underlying pms-KEM (which is not parameterized by any hash-functions) is IND-RCCA secure, then the ms-KEM is IND-RCCA secure. Note that this holds even without the restriction on nonces. This would be useful if TLS would use something like OAEP, but is currently wishful thinking.*

(2) *If the RSA and DH pms-KEM used in TLS are what they call RSA-CRE and DH-CRE secure (which say roughly that an attacker can interact with the pms-KEM only through the KEF), then the ms-KEM is IND-RCCA-N secure.*

3.3 Tolerating Weak Hash Functions

The extent to which we still have to trust MD5 ciphersuites, even if clients are configured to never negotiate a ciphersuite that uses it, is an important practical concern. Assume for instance that MD5 would be easily invertible. Then an attacker could intercept the clients encrypted *pms* in a session that is configured to use a strong hash function h and forward it in a session configured to use MD5 to the same server. Once the server starts using the master secret derived using MD5, this could now reveal information about the key derived using h.

To study the extent to which one-wayness of hash functions in H is sufficient for agile IND-RCCA-N security we define an agile variant of NR-PCA and OW-PCA security: *non-randomizability under plaintext checking oracle and key extraction function oracle attacks* (NR-PCA-KEF) and *one-wayness under plaintext checking oracle and key extraction functions oracle attacks* (OW-PCA-KEF) respectively.

Definition 5 (NR-PCA-KEF) *Consider the game below for an adversary* $\mathbf{A}$ *given oracle access to* PCA *and* KEF*:*

$pk, sk \leftarrow \mathsf{keygen}_e()$
$pv, state \leftarrow \mathbf{A}(pk)$
$c, pms \leftarrow \mathsf{enc}_e(pv, pk)$
$c' \leftarrow \mathbf{A}^{\mathsf{PCA}(\cdot,\cdot,\cdot),\mathsf{KEF}(\cdot,\cdot)}(state, c)$

$\underline{\mathsf{PCA}(pv, pms, c)}$
$pms' \leftarrow \mathsf{dec}_e(pv, sk, c)$
return $pms = pms'$

$\underline{\mathsf{KEF}(\bar{h}, \ell)}$
$\bar{h} \notin \bar{H}$ **then return** $\perp$
return $\mathsf{KEF}_{\bar{h}}(pms, \ell)$

The scheme $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e, (\mathsf{KEF}_{\bar{h}})_{\bar{h}\in\bar{H}})$ *is* $(\epsilon, t, \bar{H})$*-NR-PCA-KEF when, for any adversary* $\mathbf{A}$ *running in at most time* t*, the probability that* $pms = \mathsf{dec}_e(pv, sk, c')$ *is bounded by* ϵ*.*

Definition 6 (OW-PCA-KEF) *Consider the game below for an adversary* $\mathbf{A}$ *given oracle access to* ENC*,* DEC*, and* KEF*:*

$pk, sk \leftarrow \mathsf{keygen}_e()$
$pv, state \leftarrow \mathbf{A}(pk)$
$c, pms \leftarrow \mathsf{enc}_e(pv, pk)$
$pms' \leftarrow \mathbf{A}^{\mathsf{PCA}(\cdot,\cdot,\cdot),\mathsf{KEF}(\cdot,\cdot)}(state, c)$

$\underline{\mathsf{PCA}(pv, pms, c)}$
$pms' \leftarrow \mathsf{dec}_s(pv, sk, c)$
return $pms = pms'$

$\underline{\mathsf{KEF}(\bar{h}, \ell)}$
$\bar{h} \notin \bar{H}$ **then return** $\perp$
return $\mathsf{KEF}_{\bar{h}}(pms, \ell)$

The scheme $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e, (\mathsf{KEF}_{\bar{h}})_{\bar{h}\in\bar{H}})$ *is* $(\epsilon, t, \bar{H})$*-OW-PCA-KEF-N when, for any adversary* $\mathbf{A}$ *running in at most time* t*, the probability that* $pms = pms'$ *is bounded by* ϵ*.*

Theorem 5 (IND-RCCA-N from NR-PCA-KEF and OW-PCA-KEF KEM in the single ROM) The *ms*-KEM $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is (ϵ, t, h, H)-IND-RCCA-N for $\epsilon \leq 2\epsilon_R + \epsilon_O$ assuming that $\mathsf{KEF}_{\bar{h}}$ is a random oracle if the underlying *pms*-KEM $(\mathsf{keygen}_e, \mathsf{enc}_e, \mathsf{dec}_e)$ together with $(\mathsf{KEF}_{\bar{h}'})_{\bar{h}'\in\bar{H}\setminus\bar{h}})$ is $(\epsilon_O, t_O, \bar{H} \setminus \bar{h})$-OW-PCA-KEF secure and $(\epsilon_R, t_R, \bar{H} \setminus \bar{h})$-NR-PCA-KEF secure. Where t_O is t plus the running time of the reduction $\mathbf{B}_1$ and t_R is essentially t plus the cost of running $\mathbf{B}_2$.

The proof is very similar to Theorem 2, except that it simulates only a single random oracle and uses the $\mathsf{KEF}(.,.)$ oracle for the remaining key extraction functions.

As we model the *ms*-KEM in the random oracle model, we in fact prove security under the following definition which models a single internal key extraction function $\mathsf{KEM}_{\bar{\mathsf{h}}}$ as a random oracle:

Definition 7 (IND-RCCA-N in single KEF ROM) *Consider the game below, parameterized by* h *and* H*, using two sets of labels* Q_ℓ *and keys* Q_k *and an associative map* Q*, for an adversary* $\mathbf{A}$ *given oracle access to* ENC*,* DEC*, and* KEF*:*

$pk, sk \leftarrow \mathsf{KeyGen}_e()$
$b \leftarrow \{0,1\}$
$Q := \{\}$
$Q_\ell := \{\}$
$Q_k := \{\}$
$b' \leftarrow \mathbf{A}^{\mathsf{ENC}(\cdot),\mathsf{DEC}(\cdot,\cdot,\cdot),\mathsf{KEF}(\cdot,\cdot)}(pk)$

$\underline{\mathsf{ENC}(\ell)}$
$c, k_0 \leftarrow \mathsf{Enc}_e(h, pk, \ell)$
add k_0 *and* k_1 *to* Q_k
$k_1 \leftarrow \$$
return c, k_b

$\underline{\mathsf{DEC}(h', \ell, c)}$
if $\ell \in Q_\ell \vee h' \notin H$ **then return** $\perp$
add ℓ *to* Q_ℓ
$k \leftarrow \mathsf{Dec}_s(h', sk, \ell, c)$
if $k \in Q_k$ **then return** $\perp$
return k

$\underline{\mathsf{KEF}(\ell, pms')}$
if $Q(pms', \ell) = \perp$ **then** $Q(pms', \ell) \leftarrow \$_{\bar{h}}$
return $Q(pms', \ell)$

The scheme $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ *is* (ϵ, t, h, H)*-IND-RCCA-N in the random oracle model when, for any adversary* $\mathbf{A}$ *running in at most time* t*, the probability that* $b = b'$ *is bounded by* $\frac{1+\epsilon}{2}$.

4 Agile PRFs & Key Derivation

Let $\mathsf{Prf}_h(ms, v)$ be PRF algorithms parameterized by base algorithm descriptor h. We define the security of Prf_h as indistinguishability from a random function despite the same *ms* being also used by other algorithms $(\mathsf{Prf}_{h'})_{h' \in H \setminus h}$.

Definition 8 (Agile PRF) *Consider a game, parameterized by* h *and* H*, s.t.* $h \in H$ *for an adversary* $\mathbf{A}$ *with oracle access to* PRF*:*

$ms \leftarrow \$$
$Q := \{\}$
$b \leftarrow \{0,1\}$
$b' \leftarrow \mathbf{A}^{\mathsf{PRF}(\cdot,\cdot)}$

$\underline{\mathsf{PRF}(h', v)}$
if $h' \notin H$ **then return** $\perp$
if $h' \neq h \vee \neg b$ **then return** $\mathsf{Prf}_{h'}(ms, v)$
if $Q(v)$ *undefined* **then** $Q(v) \leftarrow \$$
return $Q(v)$

The scheme Prf_h *is* (ϵ, t, h, H)*-PRF-secure despite* $(\mathsf{Prf}_{h'})_{h' \in H \setminus h}$ *when, for any adversary running in time* t*, the probability that* $b = b'$ *is bounded by* $\frac{1+\epsilon}{2}$.

Our definition implicitly requires that the weaker algorithms Prf_h with $h \in H$ still do not leak *ms*; it omits details handled in our code, such as variable output lengths for different h.

Definition 9 (KDF & MAC for TLS) *Consider the game below using the associative maps* Q_k*,* Q_m *and parameterized by* h *and* H*, for an adversary* $\mathbf{A}$ *with oracle access to* COMMIT*,* KDF_C*,* KDF_S*,* MAC*:*

$ms \leftarrow \$$
$Q_k := \{\}; Q_m := \{\}$
$b \leftarrow \{0,1\}$
$b' \leftarrow \mathbf{A}^{\mathsf{COMMIT},\mathsf{KDF}_C,\mathsf{KDF}_S,\mathsf{MAC}}$

$\underline{\mathsf{COMMIT}(\ell, r)}$
if $\ell \in Q_k$ **then** $\perp$
$Q_k(\ell) := (\textit{committed}, r)$

$\underline{\mathsf{MAC}(h', t, v)}$
if $h' \notin H$ **then return** $\perp$
if $h' \neq h \vee \neg b$ **then return** $\mathsf{Mac}_{h'}(ms, t, v)$
if $Q_m(h', t, v)$ *undefined* **then** $Q_m(h', t, v) \leftarrow \$_m$
return $Q_m(h', t, v)$

$\underline{\mathsf{KDF}_C(\ell, h', r)}$
if $h' \notin H$ **then return** $\perp$
$k \leftarrow \mathsf{Kdf}_{h'}(ms, \ell)$
if $h' = h \wedge Q_k(\ell) = (\textit{committed}, r)$ **then**
  if b **then** $k \leftarrow \$_r$
  $Q_k(\ell) := (\textit{derived}, h', r, k)$
else $Q_k(\ell) := (\textit{done})$
return k

$\underline{\mathsf{KDF}_S(\ell, h', r)}$
if $h' \notin H$ **then return** $\perp$
$k \leftarrow \mathsf{Kdf}_{h'}(ms, \ell)$
if $h' = h \wedge Q_k(\ell) = (\textit{derived}, h', r', k')$ **then**
  if b **then**
    if $r = r'$ **then** $k := k'$**else** $k \leftarrow \$_r$
$Q_k(\ell) := (\textit{done})$
return k

The conditional key derivation scheme $(\mathsf{Kdf}_h, \mathsf{Mac}_h)$ *is* (ϵ, t, h, H) *secure despite* $(\mathsf{Kdf}_{h'}, \mathsf{Mac}_{h'})_{h' \in H \setminus h}$ *when, for any adversary* $\mathbf{A}$ *running in at most time* t*, the probability that* $b = b'$ *is bounded by* $\frac{1+\epsilon}{2}$.

Discussion. Agreeing on the agility parameter *r* as the key is derived is important for compositional proofs, and in particular to ensure that our model of the handshake fits within our model for the whole TLS protocol. Assume given a generic family of schemes $(\vec{\mathsf{Do}}_r(k, \dots))$ whose algorithms are parameterized by a key k. These schemes may provide, for instance, authenticated encryption $(\mathsf{Enc}_r(k,t), \mathsf{Dec}_r(k,c))$, or more advanced LHAE variants, such as those used in the TLS record layer of our implementation. Suppose their security is expressed using a game of the form $k \leftarrow \$; \mathbf{A}^{\vec{\mathsf{DO}}_r(k,\cdot)}$. Then, for each safely-derived key k for algorithm *r*, relying on the fact that *all* users of k will use that key with (at most) the algorithm *r*, we can create a shared instance for *r* and continue the proof with the corresponding game—provided the algorithms denoted by *r* are secure in isolation. Conversely, if both parties may start using the same fresh key k with (potentially) different algorithms r_1 and r_2, then we would need a joint, stronger, agile security assumption for these schemes.

Construction. The conditional key-derivation scheme of TLS is constructed as: $\mathsf{KDF}_h(ms, \ell) \triangleq \mathsf{PRF}_h(ms,$ `"key expansion"`$||\overline{\ell})$ and $\mathsf{MAC}_h(ms, t, v) \triangleq \mathsf{PRF}_h(ms, t||v)$, defined only for $t =$ `"client finished"` or $t =$ `"server finished"`, where $\overline{\ell}$ is ℓ after swapping the client and server random, the algorithm h is a function of a: in TLS 1.2, HMAC applied to the hash function of the ciphersuite, typically SHA256; in earlier versions of TLS, the XOR of two MD5 and SHA1 HMACs. We easily confirm the following lemma by verifying that PRF_h is used by KDF_h and MAC_h on disjoint domains.

Lemma 1 (KDF & MAC) If PRF_h is (ϵ, t, h, H)-PRF-secure, then $(\mathsf{KDF}_h, \mathsf{MAC}_h)$ is (ϵ, t', h, H)-secure, where t' is t plus a small cost for multiplexing between the functions.

From a protocol design viewpoint, more robust, modern constructions such as SP-800-108 additionally hash the target algorithm and key length for the derived key, to ensure that different algorithms always yield (computationally) independent keys. This is however not required by our definition, as it does not idealize keys in case of algorithm mismatch.

5 Defining Agile Security for the TLS Handshake

Next, we precisely define security for handshakes. Our definition is general enough to precisely capture the TLS handshake, as specified in the standard and coded in F#, while hiding most implementation details, such as message formats and cryptographic constructions.

The adversary interacts with multiple Client and Server instances of a handshake protocol Π through oracle queries. He controls when users establish "honest" long-term key pairs (KeyGen), initiate protocol instances (Init), and interact with existing instances by passing network messages (Send) and calling its control interface (Control).

Each instance has a fixed role $\mathcal{R}$, either $\mathcal{C}$ for Client or $\mathcal{S}$ for Server, and maintains a mutable status variable indicating the position in a state machine depicted in Figures 3 and 4. Each instance can go through a sequence of epochs, intuitively the number of cycles in the state machine. Each epoch records a list of *variable assigments*, extended as the adversary calls Send and triggers state transitions. Each variable is assigned at most once in every epoch. The selection and ordering of these assigments within an epoch depends on the protocol; for instance a client epoch may assign its client-certificate variable, then send a message to the server, causing the server epoch to record the same assignment later in the protocol.

Our definition focuses on these variables, which summarize what the Client and Server locally know so far about each epoch, rather than the fragments sent and received by the Handshake. We use variable assignments to express the main goals of the protocol, e.g. assigning a fresh random value to the connection-key variable *k*; and agreeing on all assignments as a session completes. We list below the main variables used in our presentation, but our definition also accounts for handshakes that provide more details (and TLS certainly does).

ℓ	epoch identifier; in TLS, the concatenation of client and server random values.
$\ell_{session}$	resumption identifier; in TLS, the identifier of the epoch that completed the session being resumed. (Our code also assigns the TLS *sessionId*, chosen by the server, but we do *not* use it as session identifier as it is not necessarily unique.)
$ext_{\mathcal{C}}$, $ext_{\mathcal{S}}$	client & server extension messages, authenticated by the handshake.
a	agility parameter; in TLS, the protocol version, the ciphersuite, and information extracted from the extension messages.
$cert_{\mathcal{C}}$, $cert_{\mathcal{S}}$	client & server certificate chains. In TLS the certificates are optional; for instance the assignment $cert_{\mathcal{C}} := \perp$ denotes the absence of client certificate.
$kex_{\mathcal{C}}$, $kex_{\mathcal{S}}$	client & server exchange variables, potentially secret, used to specify safety.
k	record key for the epoch; in TLS, depending on a, it is usually split into 4 keys for MACing & encrypting fragments in both directions.

Unless explicitly mentioned for key-exchange materials, these variables are public: the adversary can read them, but not change them; the protocol can write them once in every epoch, but not read them. (This restriction matters only for the session key k, preventing its leakage through subsequent messages once assigned a random value in Game 3 below.)

We now more precisely state the actions available to the adversary:

- $\mathsf{KeyGen}(p)$ creates and stores a new honest keypair for the long-term public-key algorithm p (ranging over s for signing and e for KEM) and returns the associated public key.
- $\mathsf{Init}(\mathcal{R}, cfg)$ creates an instance with role $\mathcal{R}$ and local configuration *cfg*; it returns a fresh handle i and a response.
- $\mathsf{Send}_i(frag)$ lets an existing instance process a fragment, depending on its current state. As a result, the instance may update its state, assign some variable, and return a response. (In TLS, responses range over sequences of Handshake and CCS fragments, intended to be sent to the peer, as well as error messages.)
- $\mathsf{Control}_i(env)$ lets the adversary influence the global, internal state of the handshake, e.g., for controlling access to stored sessions and to private keys between calls to Send. This single oracle accounts for many functions in our reference implementation.

A security model for a protocol describes how such queries are answered and how the session variables are assigned. In addition it describes two auxiliary (partial) specification functions:

- $\mathsf{pk}(cert)$ returns either the public key associated with a certificate chain (intuitively, after validating the chain), or $\perp$.
- $\mathsf{nego}(cfg, cfg')$ returns the negotiated algorithm from client and server configurations, or $\perp$ if they are incompatible.

We deliberately avoid modelling certificate chains: for the handshake, they are just bitstrings, equipped with a partial function to extract a public-key (and an underspecified description of their intended usage). Certificates are faithfully modeled, but without security guarantees, to enable the modeling of a certificate infrastructure through our application interface in future work. In particular Control provides the environment with means to reject certificates that it deems invalid.

Definition 10 (Honesty, Safety, and Completion) *For a handshake protocol* Π *and a strength predicate* $\alpha(a)$*, an adversary that calls* KeyGen*,* Init*,* Send*, and* Control *any number of times produces a trace that interleaves variable assignments for a series of epochs for each instance. In this trace:*

- *A (long-term) public key is* honest *for algorithm p if it was returned by a call to* $\mathsf{KeyGen}(p)$.
- *An epoch is* complete *when it includes the assignment complete* $:= 1$.
- *Depending on variable a, an epoch is either a* session*, with distinguished* static*,* client *and* server *exchange variables, or a* resumption*, with an $\ell_{session}$ variable.*
- *A client session is* safe *if (i) $\alpha(a)$ holds; (ii) honest keys for a's algorithms are assigned to all static exchange variables; and (iii) there is a server session with the same assignments to all server-exchange variables—and conversely for safe server sessions.*
- *A client epoch has matching algorithm $r = \mathsf{record}(a)$ when there is a server epoch with the same values for both ℓ and $\mathsf{record}(a)$.*
- *A resumption is* safe *if $\alpha(a)$ holds and $\ell_{session}$ is the identifier of a safe and complete session.*

To improve key usability, we (optionally) exclude client epochs which create the key k without agreeing with a safe server instance on the algorithms it might be used for.

Definition 11 (Handshake Game) *Let Π a handshake protocol and $\mathbf{A}$ an adversary that interacts with it by calling* KeyGen*,* Init*,* Send*, and* Control *any number of times, in any order.*

Consider the following security properties:

(1) *Uniqueness: epoch identifiers are used at most once in each role.*

Let $\mathsf{Adv}^{U}_{\Pi}(\mathbf{A})$ be the probability that two different epochs with the same role assign the same value to ℓ when $\mathbf{A}$ terminates.

(2) *Verified Safety: if the peer of a session uses a strong signature algorithm to authenticate and the public-key for the peer signature is honest, then the peer exchange value is honest.*

Let $\mathsf{Adv}^{S}_{\Pi}(\mathbf{A})$ be the probability that one epoch has the following properties when $\mathbf{A}$ terminates:

- *$\alpha(a) = 1$;*
- *the public key is honest for the signing algorithm indicated by a;*
- *the assignment to the the peer exchange value is not honest (i.e. it was not assigned by any peer);*

(3) *Agile Key Derivation: depending on a random bit b, replace the keys of safe epochs with matching algorithm r with a fresh $k \leftarrow \mathsf{KeyGen}_r()$, assigning the same value to epochs that have the same identifier ℓ, algorithms $\mathsf{kdf}(a)$ and exchange values.*

Let $\mathsf{Adv}^{K}_{\Pi}(\mathbf{A}) = 2p - 1$ where p is the probability that $\mathbf{A}$ returns b' such that $b = b'$.

(Alternatively, define Raw Key Derivation as above, except that we do not require matching rs and replace the keys with fresh random values of a fixed length.)

(4) *Agreement: for every safe & complete epoch, there is a safe epoch in the other role such that their two protocol instances agree on all prior assignments.*

Let $\mathsf{Adv}^{I}_{\Pi}(\mathbf{A})$ be the probability that the following holds when $\mathbf{A}$ terminates:

- *an instance created by $\mathsf{Create}(\mathcal{R}, cfg)$ assigns complete $:= 1$ in a safe epoch; and*
- *no instance created by $\mathsf{Create}(\overline{\mathcal{R}}, cfg')$ begins with a series of epochs with the same assignments to all variable (up to, but possibly excluding the complete assignment above).*

The handshake is (ϵ, t, α)*-secure when* $\mathsf{Adv}^{G}_{\Pi}(\mathbf{A}) \leq \epsilon$ *for all* $G = U, S, K, I$.

Discussion.

- Property (1) simply ensures that ℓ provides unique identifiers; once authenticated using (4), these identifiers can thus be used for matching client and server sessions.
- Property (2) enables, for instance, a client that trusts both the negotiated algorithm and the server certificates to infer that the server-exchange is honest, and conclude that the session is safe.
- Property (3) idealizes the derived key; this is key usability.
- Property (4) applies to all variables since the creation of the client and server instances, not just to those of the current epoch; in particular, as soon as one epoch safely completes, all prior epochs are also authenticated (even if they were not safe, or verifiably safe). This captures the enhanced guarantees provided by the safe renegotiation extension. On the other hand, the final assignment to *complete* is not itself authenticated, as the two instances asynchronously complete the epoch. Similarly, the ordering of prior assignments within matching epochs may differ.
- We give the four properties in chronological order: in TLS in particular, we first exchange fresh random values, then we derive keys, then we use them to confirm the integrity of the session negotiation. In particular, we substitute fresh random values for the derived keys before session integrity applies.
- Compatibility (nego) is a predicate on both local configurations and the instance variables, abstracting from the details of the negotiation within TLS. This prevents, for instance, algorithm regression attacks whereby the adversary can force clients and servers to negotiate weak algorithms.
- *Freshness and forward secrecy:* When a connection key is fresh and we do not require that a long-term key be honest, then we have forward secrecy. This requires dynamic compromise for signing keys and we thus give $\mathbf{A}$ an additional action $\mathsf{Corrupt}$ that returns the private key of a long-term signing key pair and marks the corresponding public key as no longer honest.

6 Proving Agile Security for the TLS Handshake

We reduce the strength for TLS to the strength of its components. Exemplary assigments performed by the TLS handshake are detailed in Figure 1.

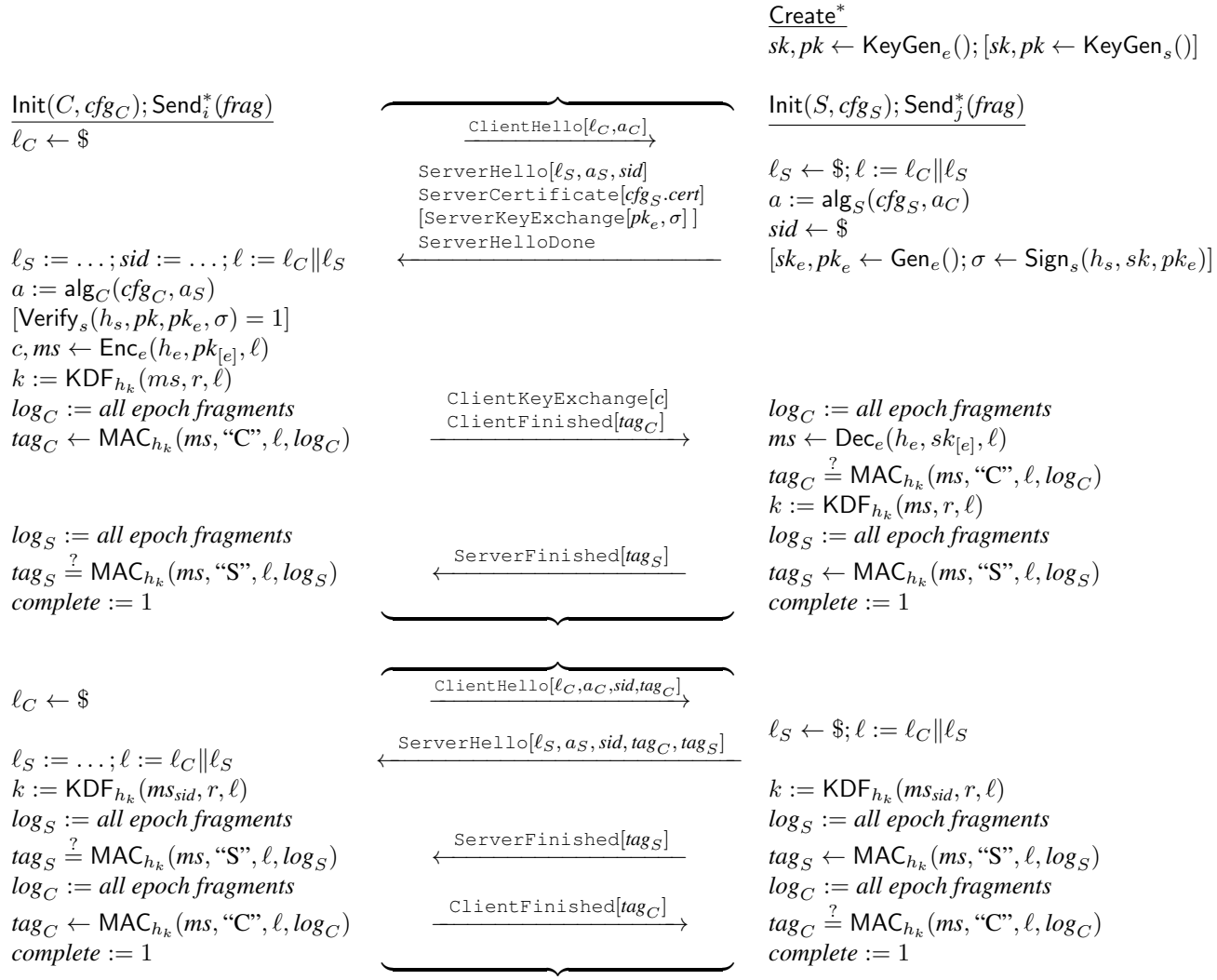
The agility parameter of the handshake a indicates which algorithm to use in each underlying functionality. We write for instance $s, h := \mathsf{sig}(a)$ for the signature and hash algorithm parameters of Definition 1.

The following theorem reduces the security of TLS to the security of its underlying algorithms, via the definition of the *strength predicate* α on agility parameters.

Theorem 6 (TLS Handshake) Let a, $a^\star$ range over the agility parameters supported by TLS, Let α be the strength predicate of Definition 10. Let n_s bound the number of calls to $\mathsf{KeyGen}(s)$. Let n and n_{ms} bound the number of epochs and sessions. Let n_e bound the number of calls to $\mathsf{KeyGen}_e()$ (counting both ephemeral and static KEMs). The TLS handshake is (ϵ, t, α)-secure under the following assumptions:

(1) Let $H = \{h^\star \mid s, h^\star := \mathsf{sig}(a^\star)\}$.
If $\alpha(a)$ and $s, h := \mathsf{sig}(a)$, then $(\mathsf{KeyGen}_s, \mathsf{Sign}_s, \mathsf{Verify}_s)$ is $(\epsilon_{s,h}, t_{s,h}, h, H)$-secure.

(2) Let $H = \{h^\star \mid e, h^\star := \mathsf{kem}(a^\star)\}$.
If $\alpha(a)$ and $e, h := \mathsf{kem}(a)$, then $(\mathsf{KeyGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is $(\epsilon_{e,h}, t_{e,h}, h, H)$-IND-RCCA.



We omit variable assignments copying elements from the messages. We let a abbreviate all negotiated algorithms, i.e. e, h_e, s, h_s, h_k … We omit the two CCS messages, telling TLS to start using the new connection algorithms and keys, as they play no role in the handshake (and are not included in its fragment log).

For a given role, "all epoch fragments" means the concatenation of all fragments it has sent and received at this point of the protocol, starting from the latest `ClientHello` message, which marks the beginning of the current epoch. Thus, the second log_S includes a MAC of the first log_C as part of the `ClientFinished` message.

The integrity of sequences of epochs is guaranteed via the (mandatory) SecureRenegotiation extension in the ClientHello and ServerHello messages, which causes the two *tag* values of the prior epoch to be included in both *tag* values for the current epoch.

Figure 1: Outline of the Handshake protocol, showing (1) the static Handshake protocol parameterized by KEM (Gen, Enc, Dec) and conditional KDF (KDF, MAC), followed by (2) a resumption.

(3) Let $H = \{h^\star \mid h^\star := \mathsf{prf}(a^\star)\}$.
If $\alpha(a)$ and $h := \mathsf{prf}(a)$, then $(\mathsf{Kdf}_h, \mathsf{Mac}_h)$ is (ϵ_h, t_h, h, H)-secure.

where $\epsilon = \sum_s (n_s \sum_h \epsilon_{s,h}) + \sum_e n_e (\sum_h \epsilon_{e,h}) + n_{ms} (\sum_h \epsilon_h) + n^2 (2^{-225} + 2^{-\mathsf{min}_h |\mathsf{Mac}_h|})$ and each t_* in the assumptions is at most t plus the cost of simulating Π in the reduction.

In the statement, the last two contributions to ϵ account for the probability of a collision between any two client nonces, any two server nonces, and any two finished-message tags (two for each local epoch).

In combination with Bhargavan et al. [2013], we get crypto-agile application security, under those assumptions for the handshake and their assumptions for the transport layer algorithms. Here, we require that connection keys can be used with any non-agile authenticated encryption. We achieve this as we do not provide security for safe epochs without matching algorithms as they will never complete.

We look both at full (sessions) and abbreviated handshakes (resumptions) at once as large parts of the proof and the bounds are shared.

PROOF: We define the client exchange value to be the master secret *ms* and the KEM public key pk_e, and the server exchange value to be the public key pk_e of the KEM. The latter can be either static or ephemeral.

(1) Uniqueness. Let n be the total number of epochs. The length of the randomness in client and server nonces is 224 bits. The probability that n randomly generated 224 bit values give rise to a collision is approximately $\binom{n}{2} 2^{-224}$. This is the worst case as it assumes that the adversary controls half of ℓ and that all of them are of the same role. We thus bound $\mathsf{Adv}^U_\Pi(\mathbf{A})$ by $n^2 2^{-225}$. This proves uniqueness both for sessions and resumptions.

(2) Verified Safety. We need to show that if the signing algorithm is strong and the public-key for the peer's signature is honest, then there is a peer session with the same assignments to all (ephemeral) peer-exchange variables.

For anonymous peers there is nothing to proof. They do not have public keys and their communication partners cannot verify the safety of their session. Conversely, the servers of static cipher-suites like RSA and static Diffie-Hellman have only static server exchange values.

For clients using ephemeral Diffie-Hellman, the client verifies that the server's ephemeral DH contribution is signed.[6] Servers with authenticated clients verify a signature on the clients transcript up to sending the `ClientKeyExchange` fragment. Only these last two cases require a reduction proof.

The proof is by a series of hybrids. Note that our definition allows the same signature keys to be used both by client and server epochs, we thus consider hybrids that consider both cases at once.

- *Game 1* is the original verified safety game, in which $\mathbf{A}$ interacts with the TLS handshake protocol by calling KeyGen, Init, Send, and Control any number of times, in any order.

- *Game 2* is the same as Game 1, except that we abort the game (without $\mathbf{A}$ winning) if a client instance receives an ephemeral KEM key signed by a strong signature scheme with an honest signing key, that has not been assigned in a server instance variable; or if a server instance receives such a strong signature in a `CertificateVerify` message such that there does not exist a client epoch for which the values authenticated by the signature on the transcript have been assigned.

  Let n_s be the number of signing keys for signature algorithm s used by the server and let the other constants be as defined in the theorem. We have that $|\mathsf{Adv}^{G1}_\Pi(\mathbf{A}) - \mathsf{Adv}^{G2}_\Pi(\mathbf{A})| \leq \sum_s (n_s \sum_h \epsilon_{s,h})$.

  To achieve this property we have to prove that client and server signatures cannot be confused. Then the server signature guarantees the honesty of the server exchange value and the authenticated values of the client signature uniquely determine the master secret and the KEM public key at the client which must thus also have been derived and assigned in the same client session.

[6] Exotic ciphersuites also support ephemeral variants of RSA key transport. This could be modeled in a similar fashion. But is not supported by our implementation.

PROOF: We show how to use an adversary $\mathbf{A}$ for which $|\mathsf{Pr}^{G1}_{\Pi}(b'=1) - \mathsf{Pr}^{G2}_{\Pi}(b'=1)| > \sum_s (n_s \sum_h \epsilon_{s,h})$ to break EUF-CMA security of the signature scheme. The proof is via a hybrid argument over the honest signature public keys with strong algorithm.

We describe Hybrid (i,j). For public keys up to i, it performs the checks and aborts of *Game 2*. For key i with $h \leq j$ (for some ordering on the hash algorithms) it also does these aborts. For key i but $h > j$ and public keys greater than i it behaves like in *Game 1*.

Let s be the public key algorithm of the signature key i. We describe a reduction $\mathbf{B}_{i,j}$ that uses an $\epsilon_{s,j}$ difference in the advantage of $\mathbf{A}$ between two hybrids to break EUF-CMA security. For public key i the reduction uses the public key from the EUF-CMA game. The reduction uses calls to Sign to sign using the corresponding private key. Until $\mathbf{A}$ produces a forgery for key i and hash algorithm j, the reduction $\mathbf{B}_{i,j}$ behaves exactly like Hybrid $(i,j-1)$ (Hybrid (i, h_{max}) or Hybrid $(i-1, h_{min})$ at key borders). $\mathbf{B}_{i,j}$ simply forwards a forgery to win the EUF-CMA game.

As we sum over all hybrids, we thus have that $|\mathsf{Adv}^{G1}_{\Pi}(\mathbf{A}) - \mathsf{Adv}^{G2}_{\Pi}(\mathbf{A})| \leq \sum_s (n_s \sum_h \epsilon_{s,h})$.

In Game 2, $\mathsf{Adv}^{G2}_{\Pi}(\mathbf{A}) = 0$ and thus $\mathsf{Adv}^{S}_{\Pi}(\mathbf{A}) \leq \sum_s (n_s \sum_h \epsilon_s)$. Verified safety applies only to sessions.

(3) Agile Key Derivation. The proof proceeds using a sequence of games. Let $\mathsf{Pr}^{Gi}_{\Pi}(b'=1)$ be the probability that $\mathbf{A}$ outputs 1 in Game i.

- *Game 1*. This is the agile key derivation game for $b = 0$.
- *Game 2*. This is the same as Game 1, except that we abort if there are colliding ℓ.
  $|\mathsf{Pr}^{G1}_{\Pi}(b'=1) - \mathsf{Pr}^{G2}_{\Pi}(b'=1)| \leq \mathsf{Adv}^{U}_{\Pi}(\mathbf{A})$.
- *Game 3*. The same as Game 1 except that we replace the *ms* output of the KEM Enc and Dec algorithms in safe epochs with random values. It assigns the same random value to epochs that have the same identifier ℓ, $\mathsf{kdf}(a)$, and exchange values.
  $|\mathsf{Pr}^{G2}_{\Pi}(b'=1) - \mathsf{Pr}^{G3}_{\Pi}(b'=1)| \leq \sum_e n_e (\sum_h \epsilon_{e,h})$.

  PROOF: We show how to use an adversary $\mathbf{A}$ for which $|\mathsf{Pr}^{G2}_{\Pi}(b'=1) - \mathsf{Pr}^{G3}_{\Pi}(b'=1)| > \sum_e n_e (\sum_h \epsilon_{e,h})$ to break KEM security. The proof is via a hybrid argument over the different KEM keys with strong algorithm.

  We describe Hybrid (i,j). For public keys up to i, it uses KEMs with random master secrets. For key i with $h \leq j$ (for some ordering on the hash algorithms) it uses random master secrets; for $h > j$ it uses concretely generated master secrets. For public keys greater than i, it uses KEMs with concretely generated master secrets.

  Let e be the public key algorithm of the KEM key i. We describe a reduction $\mathbf{B}_{i,j}$ that uses an $\epsilon_{e,j}$ difference in the probabilities of $\mathbf{A}$ between two hybrids to break KEM security. For public key i, the $\mathbf{B}_{i,j}$ uses the public key from the KEM game. For hash algorithms j, it calls to ENC to obtain the *ms* returned by the KEM Enc algorithm. We store it in a table indexed by a_{KDF}, identifier ℓ, and exchange values. For other hash algorithms it runs concrete KEM encryption.

  The reduction uses calls to DEC to replace the *ms* returned by the KEM Dec algorithm. In case DEC returns $\perp$ we return the key that was provided by the ENC oracle call with the same index for the table.

  The reduction behaves exactly like Hybrid (i,j) or Hybrid $(i,j-1)$ (Hybrid (i, h_{max}) or Hybrid $(i-1, h_{min})$ at key borders) based on the internal bit b of the KEM game. $\mathbf{B}_{i,j}$ simply forwards the guess of A.

  As we sum over all hybrids, we thus have that $|\mathsf{Pr}^{G2}_{\Pi}(b'=1) - \mathsf{Pr}^{G3}_{\Pi}(b'=1)| \leq \sum_e n_e (\sum_h \epsilon_{e,h})$. □

- *Game 4*. Game 4 is the same as Game 3, except that we sample fresh keys (in exactly the same way as in the $b = 1$ branch.

  $|\mathsf{Pr}^{G3}_{\Pi}(b' = 1) - \mathsf{Pr}^{G4}_{\Pi}(b' = 1)| \leq n_{ms}(\sum_h \epsilon_h)$.

  PROOF: We show how to use an adversary $\mathbf{A}$ for which $|\mathsf{Pr}^{G3}_{\Pi}(b' = 1) - \mathsf{Pr}^{G4}_{\Pi}(b' = 1)| > n_{ms}(\sum_h \epsilon_h)$ to break conditional KDF security. The proof is via a hybrid argument over the (strongly) safe KDF keys *ms* and their strong algorithms.

  We describe Hybrid (i, j). For *ms* up to i, it uses KDFs that sample keys according to $\mathsf{KeyGen}_r()$ and random MAC tags (idealized output). For key i with $h \leq j$ it provides idealized output; for $h > j$ it uses concretely generated keys and MAC tags. For master secrets greater than i, it uses KDFs with concretely generated keys and tags.

  We now describe a reduction $\mathbf{B}_{i,j}$ that uses an ϵ_j difference in the advantage of $\mathbf{A}$ between two hybrids to break conditional KDF security. For master secret i, $\mathbf{B}_{i,j}$ uses the conditional KDF game. It calls $\mathsf{COMMIT}(\ell, \mathsf{a})$ once a server assigned both the ℓ and a variables. It calls KDF_C to obtain the keys of client epochs and KDF_S to obtain the keys of server epochs. The reduction uses calls to MAC to obtain MAC tags for both client and server finished messages.

  The reduction behaves exactly like Hybrid (i, j) or Hybrid $(i, j-1)$ (Hybrid (i, h_{max}) or Hybrid $(i-1, h_{min})$ at master secret borders) based on the internal bit b of the conditional KDF game. $\mathbf{B}_{i,j}$ simply forwards the guess of A.

  As we sum over all hybrids, we thus have that $|\mathsf{Pr}^{G3}_{\Pi}(b' = 1) - \mathsf{Pr}^{G4}_{\Pi}(b' = 1)| \leq n_{ms}(\sum_h \epsilon_h)$. □

- *Game 5-Game 6*. The same as Game 4, except that in Game 5 we revert to generating master secrets for the $b = 0$ branch as in Game 2; and in Game 6 we revert to allowing collisions on ℓ.

Game 5 behaves just like the agile key derivation game for $b = 1$ so,

$$\mathsf{Adv}^{K}_{\Pi}(\mathbf{A}) \leq |\mathsf{Pr}^{G1}_{\Pi}(b' = 1) - \mathsf{Pr}^{G6}_{\Pi}(b' = 1)| \leq 2 \cdot \left(\mathsf{Adv}^{U}_{\Pi}(\mathbf{A}) + \sum_e n_e(\sum_h \epsilon_{e,h}) \right) + n_{ms}(\sum_h \epsilon_h) \; .$$

Resumptions do not require Game 3 and Game 5, as the master secret is reused from the resumed session.

Raw Key Derivation. The protocol in Figure 1 does not meet the *raw key derivation* property, if KDF returns different keys for different record algorithms as indeed the case in TLS as keys are cut to the required length. If it is changed to return constant size keys, then it becomes secure. The proof is similar to the proof above, except that the reduction calls COMMIT for both the client and the server with an a with a constant record algorithm. Note that *Agile Key Derivation* is not sufficient for providing guarantees for *false start* as it guarantees that the same record keying material will never be used with different record algorithms. Instead, *false start* requires *Raw Key Derivation* security for the handshake and stronger agile security properties for record algorithms that may share raw keys.

(4) Agreement. The proof proceeds using a sequence of games.

- *Game 1-Game 4* are same as the corresponding games for agile key derivation.

  $|\mathsf{Adv}^{G1}_{\Pi}(\mathbf{A}) - \mathsf{Adv}^{G2}_{\Pi}(\mathbf{A})| \leq |\mathsf{Pr}^{G1}_{\Pi}(b' = 1) - \mathsf{Pr}^{G4}_{\Pi}(b' = 1)|$

  To see this consider the attacker $\mathbf{A}'$ on those games in the agile key derivation proof which output 1 whenever $\mathbf{A}$ succeeds in breaking *agreement*.

As in Game 4 MACs of safe epochs are generated at random, $\mathsf{Adv}^{G4}_{\Pi}(\mathbf{A}) \leq 2 \cdot (\binom{n}{2} 2^{-\mathsf{min}_h|\mathsf{Mac}_h|}) \leq 2^{-\mathsf{min}_h|\mathsf{Mac}_h|}$ by the collision probability of MAC tags.

Recall that the safe renegotiation extension requires that the log includes the tag of the log of prior epochs which means that we authenticate all assignment up to the current epoch and thus

$$\begin{aligned}\mathsf{Adv}^{I}_{\Pi}(\mathbf{A}) &\leq |\mathsf{Pr}^{G1}_{\Pi}(b' = 1) - \mathsf{Pr}^{G4}_{\Pi}(b' = 1)| + 2^{-\mathsf{min}_h|\mathsf{Mac}_h|} \\ &\leq \mathsf{Adv}^{U}_{\Pi}(\mathbf{A}) + \sum_e n_e(\sum_h \epsilon_{e,h}) + n_{ms}(\sum_h \epsilon_h) + 2^{-\mathsf{min}_h|\mathsf{Mac}_h|} \ .\end{aligned}$$

Resumptions do not require Game 3, as the master secret is reused from the resumed session. □

7 Code-Based Verified Implementation

Our cryptographic proofs apply to an interoperable reference implementation of the TLS Handshake written in the F# programming language. Here, we give salient details of the implementation and describe how its verification relates to the cryptographic models of the preceding sections. Where possible we highlight how implementation details motivate our cryptographic proofs. More details on the implementation are presented in the Appendix.

Our implementation is around five thousand lines of code and supports four protocol versions, six key exchange mechanisms, six encryption algorithms, and four hash functions, as listed in Table 1.

Our TLS library consists of forty five modules, not counting application code or platform libraries, as depicted in Figure 5. Each module implements a single cryptographic functionality or protocol component and represents an abstraction boundary through its interface. A module is either trusted to be implemented correctly (e.g. the session database), or idealized under a cryptographic assumption (e.g. Diffie-Hellman groups), or is verified to meet its interface. The interface specifies preconditions, postconditions, and type abstraction rules that govern the conditions under which secrets (such as encryption keys and plaintexts) may be read or written by a module.

We use the type-based verification method of **??**. Modules implementing core cryptographic primitives are replaced by idealization, supported by documented cryptographic assumptions. They are then given typed interfaces that encapsulate both the conditions under which they may be safely used and the guarantees they then provide. The rest of the implementation is then verified along with these idealizations against application-level security goals.

We highlight three aspects of our implementation before presenting perfomance evaluation results.

7.1 *Control* Interface

The application can control the TLS client and server by calling functions in the TLS module, which in turn calls the relevant functions in the Handshake module. The main functions in this interface are:

val *init*: *rl*:*Role* → *c*:*config* → (*ci*:*CI* ∗ *s*:(;*ci*)*state*){*Config*(*ci*,*s*) = *c* ...}
val *authorize*: *r*:*Role* → *si*:*SessionInfo* → unit {*Authorize*(*r*,*si*)}
val *resume*: *nextSID*:*sessionID* → *c*:*config* → (*ci*:*CI* ∗ *s*:(;*ci*)*state*){ *Config*(*ci*,*s*) = *c* ...}

7.2 Message Formats

After initialization, the Handshake module listens to messages from the network, which represent the adversary's *Send* interface. It parses each message and then calls the relevant function to modify the Handshake state and adds the message to the Handshake log for eventual authentication in the Finished (and Client CertificateVerify) messages.

Detailed message formats are traditionally ignored in protocol models and cryptographic proofs, but are crucial in TLS to establish *Agreement*, which depends on the both the client and server having the same parsed

interpretation of their Handshake message logs. To give an example, the first message in the log, the ClientHello has the following format:

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16−2>;
    CompressionMethod compression_methods<1..2^8−1>;
    select (*extensions_present*) {
        case **false**:
            struct {};
        case **true**:
            Extension extensions<0..2^16−1>;
    };
} *ClientHello*;

To ensure that this message can be parsed unambiguously at both client and server, we define a logical function *ClientHelloMsg*(*pv*,*crand*,*sid*,*cs*,*cl*,*ext*) that precisely details this message format. We then prove that our code for generating and parsing client hello messages obeys this function:

val *clientHelloBytes*: *c*:*config* → *cr*:*random* → *sid*:*sessionID* → *ext*:bytes → *m*:bytes{*B*(*m*) = *ClientHelloMsg*(*c*.*maxVer*,*cr*,*sid*,*c*.*ciphersuites*,*c*.*compressions*,*ext*)}

Then, we prove that the logical function is injective, so that there is a unique way to parse its components.

theorem !*pv*,*crand*,*sid*,*cs*,*cl*,*ext*,*pv*',*crand*',*sid*',*cs*',*cl*',*ext*'.
ClientHelloMsg(*pv*,*crand*,*sid*,*cs*,*cl*,*ext*) = *ClientHelloMsg*(*pv*',*crand*',*sid*',*cs*',*cl*',*ext*') ⇔ (*pv* = *pv*' ∧ *crand* = *crand*' ∧ ...)

Finally, we extend this injectivity theorem to the handshake log. Any two equal logs must begin with the same client hello message, and hence with the same hello parameters.

7.3 State Machine

The bulk of the protocol logic is encoded in the handshake state machine, as depicted in Figures 3 and 4. Encoding and verifying such a complex state machine is a challenge—not only does it have to implement different congtrol flow paths for different key exchanges, different protocol versions, and client authentication modes, it must also be ready to receive messages that trigger any handshake at any time.

In such code, it is easy to make some implementation decisions that end up bypassing security. For example, if a client renegotiates a full handshake with the server, then during this second handshake it may contine to receive data over the connection established from the first handshake. It should accept this data until it receives the new ChangeCipherSpec message, at which point it should stop accepting data until the Handshake is complete, since new keys have been committed on but not confirmed. However, many TLS implementations make the mistake of accepting data even in this inconsistent state. Our implementation carefully enforces such state machine invariants.

As a second example, suppose a client has sent its Finished message and is waiting for the server's Finished message. It is tempting for the client to start sending its data already to reduce the latency of the TLS connection. This is the design of the TLS False Start extension, and a similar rationale is used in the TLS NPN NextProtocolMessage. In both cases, if the ciphersuite negotiated is strong enough, the confidentiality of the data being sent seems to be preserved. But cryptographically, it is difficult to justify a design where a client and server may use the same keys with different algorithms. Moreover, we found several conditions where such encrypted data

Ciphersuite			F# (BC)		F# (EVP)		OpenSSL		Oracle JSSE	
KEX	Enc	MAC	HS/s	MiB/s	HS/s	MiB/s	HS/s	MiB/s	HS/s	MiB/s
RSA	RC4	MD5	268.22	43.44	273.81	89.54	1257.50	255.99	410.55	64.59
RSA	RC4	SHA	272.32	38.13	270.84	84.76	1214.58	216.20	419.67	59.47
RSA	3DES	SHA	259.86	8.54	272.32	18.82	1147.40	22.12	383.58	10.47
RSA	AES128	SHA	266.23	22.84	269.96	50.10	1121.55	261.74	406.55	58.84
RSA	AES128	SHA256	268.80	19.37	271.13	43.12	1121.56	122.36	401.56	47.87
RSA	AES256	SHA	261.77	20.11	271.13	41.21	1185.66	221.06	-	-
RSA	AES256	SHA256	257.45	17.39	270.84	35.94	1087.29	111.88	-	-
DHE	3DES	SHA	20.83	8.46	20.96	18.32	336.92	22.19	-	-
DHE	AES128	SHA	21.02	22.69	20.85	47.72	343.43	277.64	-	-
DHE	AES128	SHA256	20.94	19.16	20.84	43.46	338.76	123.19	-	-
DHE	AES256	SHA	20.56	20.12	20.95	40.04	344.86	246.14	-	-
DHE	AES256	SHA256	21.11	17.62	20.79	35.69	339.22	113.37	-	-

Figure 2: Performance benchmarks (OpenSSL 1.0.1e as server).

may be sent too optimistically, and may be leaked to the network-based adversary. Our implementation strictly forbids such early data transmission.

We verify our state machine preserves its logical invariants; this proof is for a fifteen hundred line program and requires the use of an SMT solver. We also verify that the state machine treats all secrets parameterically, as a precondition to the game-based transformations of earlier sections.

7.4 Performance Evaluation

We evaluate the performance of our implementation, written in F# and linked to the Bouncy Castle C# cryptographic and the OpenSSL EVP cyrptographic providers, against two popular TLS implementations: OpenSSL 1.0.1e, written in C and using its own aforementioned cryptographic libraries (EVP), and Oracle JSSE 1.7, written in Java and using the SunJSSE cryptographic provider.

We tested clients and servers for each implementation against one another, running on the same host to minimize network effects. Figure 2 reports our results for different clients and ciphersuites with OpenSSL as server. We measured (1) the number of Handshakes completed per second; and (2) the average throughput provided on the transfer of a 400 MB random data file.

At first glance, when comparing to OpenSSL, these results highlight that our reference implementation has been designed primarily for modular verification, and has not been optimized for speed. For example, all buffers are implemented using plain functional byte arrays which involve a lot of dynamic allocation and copying as record fragments are processed. However, when compared to VM-based languages, the slow-down is less prominent (order of magnitude of 2 for the JSSE case), and we consistently outperforms the rudimentary TLS client distributed with Bouncy Castle. Moreover, when changing our crypto provider from BouncyCastle to OpenSSL EVP, one can notice that our throughput is then 1.5 faster in our implementation than in the JSSE case.

Table 1: Supported protocol versions, ciphersuites and extensions.

Protocol Versions	Key exchange	Record encryption	Record HMAC	Extensions
TLS 1.2	RSA	AES_256_GCM	SHA384	Secure renegotiation
TLS 1.1	DHE_DSS	AES_128_GCM	SHA256	
TLS 1.0	DHE_RSA	AES_256_CBC	SHA	
SSL 3	DH_RSA	AES_128_CBC	MD5	
	DH_DSA	3DES_EDE_CBC		
	DH_anon	RC4_128		

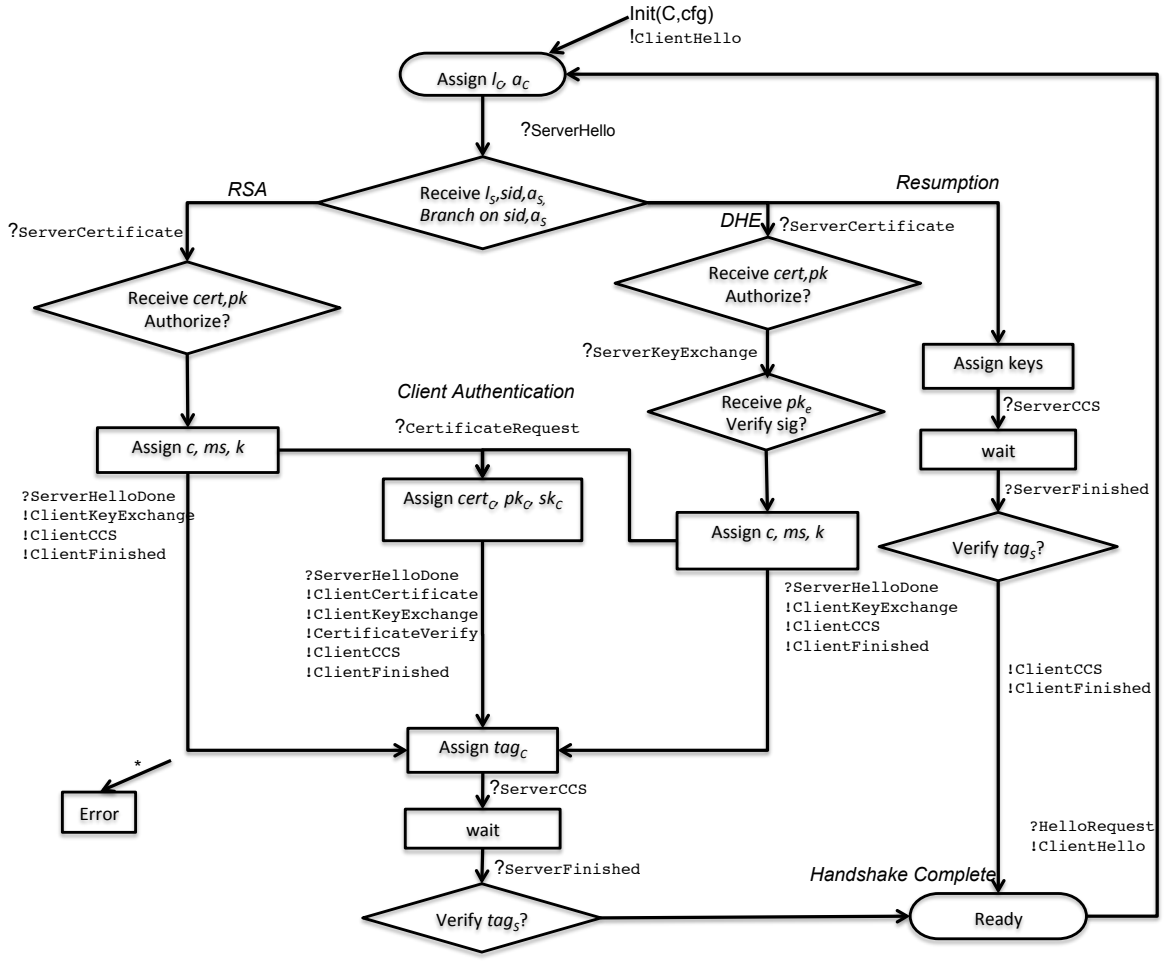


Figure 3: State machine of the client handshake module.

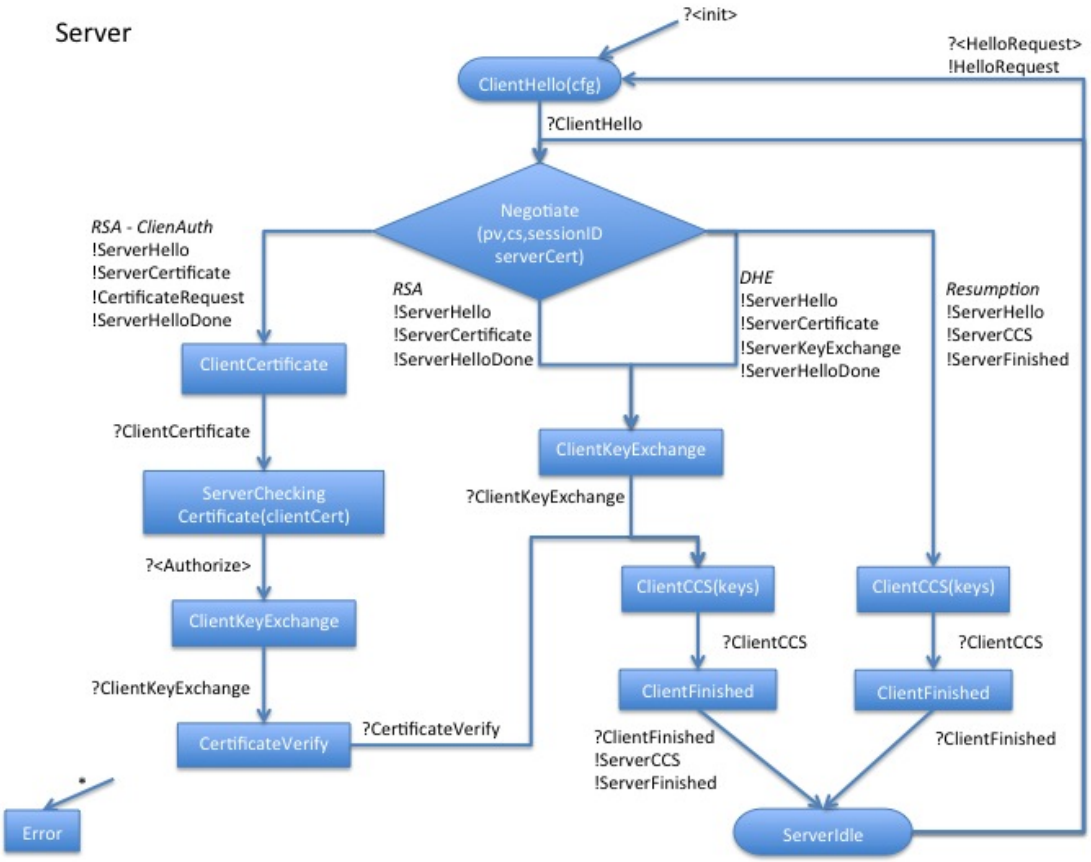


Figure 4: State machine of the server handshake module.

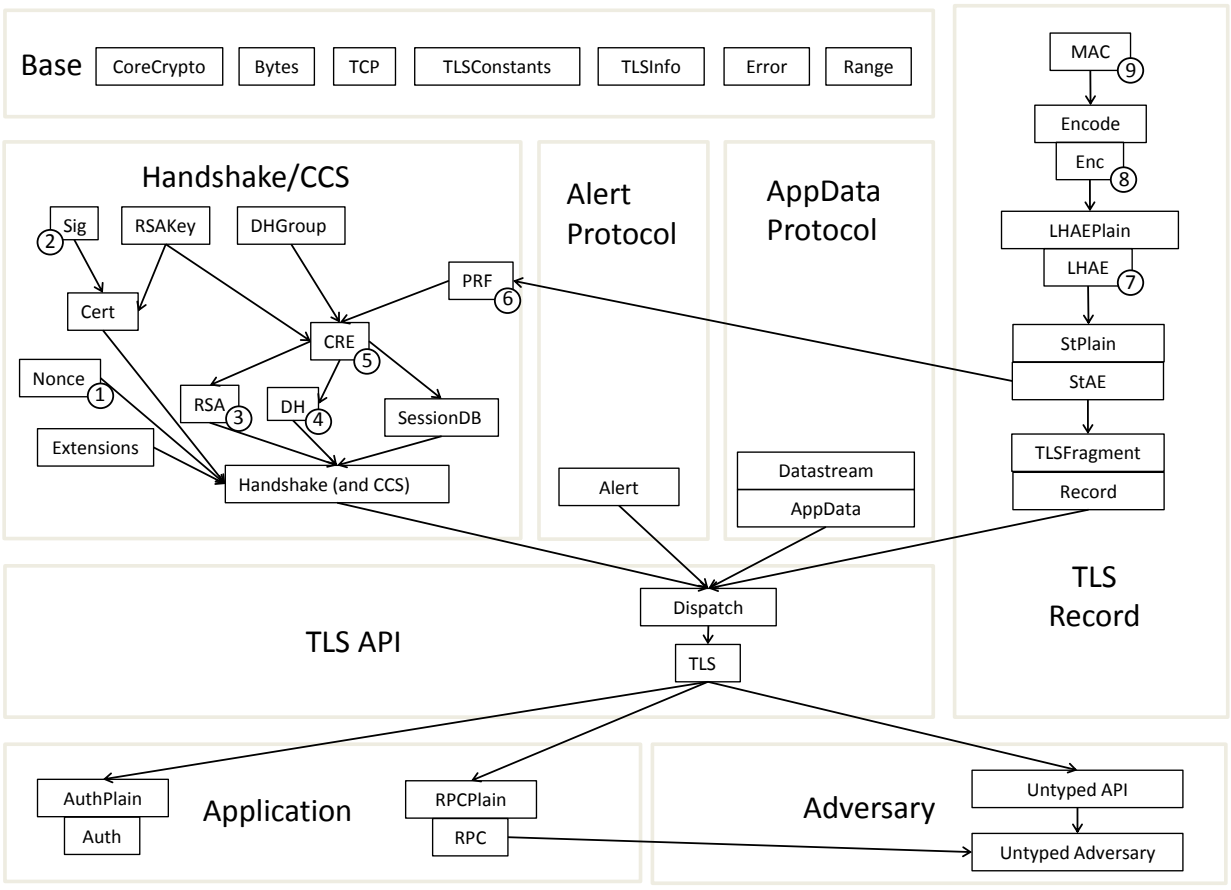


Figure 5: Modular implementation of TLS

References

T. Acar, M. Belenkiy, M. Bellare, and D. Cash. Cryptographic agility and its relation to circular encryption. In *EUROCRYPT*, pages 403–422, 2010.

R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer, 2012.

G. Barthe, D. Pointcheval, and S. Z. Béguelin. Verified security of redundancy-free encryption from rabin and rsa. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 724–735. ACM, 2012. ISBN 978-1-4503-1651-4.

M. Bellare and P. Rogaway. Entity authentication and key distribution. In *CRYPTO*, pages 232–249, 1993.

K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptopgrahic security. In *IEEE Symposium on Security and Privacy (SP'2013)*, pages 445–469, 2013.

D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *CRYPTO'98*, pages 1–12, 1998.

B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In *CT-RSA*, 2011.

C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi, and S. C. Williams. Less is more: Relaxed yet composable security notions for key exchange. *IACR Cryptology ePrint Archive*, 2012:242, 2012.

R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In *EUROCRYPT*, pages 337–351, 2002.

R. Canetti, H. Krawczyk, and J. B. Nielsen. Relaxing chosen-ciphertext security. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2003.

R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM Conference on Computer and Communications Security*, pages 49–60. ACM, 2013. ISBN 978-1-4503-2477-9.

M. Fischlin, A. Lehmann, and D. Wagner. Hash function combiners in TLS and SSL. In J. Pieprzyk, editor, *CT-RSA 2010*, volume 5985 of *LNCS*, pages 268–283. Springer, Mar. 2010.

P.-A. Fouque, D. Pointcheval, and S. Zimmer. HMAC is a randomness extractor and applications to TLS. In *ASIACCS*, pages 21–32, 2008.

S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk. Universally composable security analysis of TLS. In *ProvSec*, pages 313–327, 2008.

M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS*, pages 38–49, 2012.

F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. Cryptology ePrint Archive, Report 2012/630, 2012. http://eprint.iacr.org/.

D. Hofheinz and E. Kiltz. Secure hybrid encryption from weakened key encapsulation. In *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 553–571. Springer, 2007. ISBN 978-3-540-74142-8.

T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, pages 273–293, 2012.

J. Jonsson and B. S. Kaliski. On the security of RSA encryption in TLS. In *CRYPTO*, pages 127–142, 2002.

V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, pages 426–440, 2003.

H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer, 2013. ISBN 978-3-642-40040-7.

R. Küsters and M. Tuengerthal. Composition theorems without pre-established session identifiers. In *CCS*, pages 41–50, 2011.

A. Langley. Unfortunate current practices for HTTP over TLS, 2011. http://www.ietf.org/mail-archive/web/tls/current/msg07281.html.

N. M. Langley, A. and B. Moeller. Transport Layer Security (TLS) False Start. Internet Draft, 2010.

N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *CCS*, pages 62–72, 2012.

P. Morrissey, N. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. In *ASIACRYPT'08*, pages 55–73, 2008.

T. Okamoto and D. Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In K. Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2001. ISBN 3-540-41658-7.

K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*

2011, pages 372–389, 2011.

M. Ray. Authentication gap in TLS renegotiation. http://extendedsubset.com/Renegotiating_TLS.pdf, 2009.